面向程序员的数据挖掘指南

wizardforcel

Published with GitBook



目錄

面向程序员的数据挖掘指南

> 作者：Ron Zacharski
>
> 原文：A Programmer's Guide to Data Mining
>
> 来源：guidetodatamining
>
> 协议：CC BY-NC 3.0

欢迎辞



这是一本用于学习基本数据挖掘知识的书籍。大部分关于数据挖掘的书籍都着重于讲解理论知识，难以理解，让人望而却步。不要误会，这些理论知识还是非常重要的。但如果你是一名程序员，想对数据挖掘做一些了解，一定会需要一本面向初学者的入门书籍。这就是撰写本书的初衷。

这本指南采用“边学边做”的方式编写，因此在阅读本书时，我强烈建议您动手实践每一章结束提供的练习题和实验题，使用书中的Python脚本将其运行起来。书中有一系列展示数据挖掘技术的实例，因此在阅读完本书后，你就能掌握这些技术了。这本书以Creative Commons协议发布，可以免费下载。你可以任意分发这本书的副本，或者重新组织它的内容。也许将来我会提供一本纸质的书籍，不过这里的在线版本永远是免费的。

目录

这本书以PDF格式免费发行，点击下面每一章的标题，您就会被定向到一个页面，其中包含了这一章的PDF文件和示例代码的下载链接。此外，你还可以在这个页面中发表评论，举出书中的问题和错误，哪部分难以理解等。我会根据这些评论来修订本书。

第一章：简介

讲述什么是数据挖掘，它所能解决的问题的是什么，以及在阅读完本书后，你可以做些什么。

第二章：推荐系统入门

介绍协同过滤，基本的距离算法，包括曼哈顿距离、欧几里得距离、闵科夫斯基距离、皮尔森相关系数。使用Python实现一个基本的推荐算法。

第三章：隐式评价和基于物品的过滤算法

这章开始讨论可供选择的用户评价体系。用户能够显示地給于评价（好、差、五星评价等），或者隐式地給于评价——如果用户在亚马逊购买了一个MP3，我们则认为他是“喜欢”这件商品的。

第四章：分类

上一章中我们使用用户对商品的评价来进行推荐，这一章我们会使用商品本身的特性来进行推荐。这种算法在潘多拉等网站中采用。

第五章：进一步探索分类

本章会讨论如何评价分类器的效果，方法包括十折交叉验证、留一法、以及Kappa检验等，同时还会引入kNN算法。

第六章：朴素贝叶斯

我们会在这章探索朴素贝叶斯分类算法，使用概率密度函数来处理数值型数据。

第七章：朴素贝叶斯算法和非结构化文本

这一章我们会尝试使用朴素贝叶斯算法来对非结构化文本进行分类。我们是否能够判断出Twitter上的一片影评是正面评价还是负面的呢？

第八章：聚类

我们会讨论层次聚类和kmeans聚类。

第一章：简介

原文链接：http://guidetodatamining.com/chapter-1/

内容：

- 寻找事物
- 本书结构
- 阅读完本书后你可以做些什么？
- 为什么数据挖掘很重要？哪些内容可以为我所用？
- 标题里的“Numerati的古老艺术”是什么意思？

序



如果你每天都能重复做这些简单的事，你就会获得某种特别的力量。在你获得之前，这是特别的，但获得之后，就没什么大不了的了。

——鈴木 俊隆

在阅读本书之前，你可能会认为像潘多拉、亚马逊那样的推荐系统、或是恐怖分子用的数据挖掘系统，一定会非常复杂，只有拥有博士学位的人才能够了解其中的算法。你也许会认为设计出这些系统的人都是研究火箭技术的。而我撰写本书的目的之一就是希望能够揭开这些系统的神秘面纱，展示它们所使用的基本原理。虽然的确会有像Google工程师或是在国家安

全局工作的天才技术人员，数据挖掘却是建立在一些基本逻辑和方法之上的，非常易于理解。在阅读本书之前，你可能会认为数据挖掘是一种让人震惊的技术，但阅读之后你会发现，其实也没什么大不了的。

上图中的日本文字“初心”，表示要始终保持一颗“初学者的心”，也就是一种开放的心态，接受各种可能性。下面这个故事你可能在哪儿听过（很有可能是来自李小龙的“龙争虎斗”）：一位教授想要寻求指引，于是来到一位智者面前，希望能得到点化。这个教授不停地说着自己毕生学到了什么，发表了多少论文等等。这时，智者问他：“喝茶吗？”然后开始向教授的杯子里倒茶，一直倒，最后溢到了桌子上、地上。“你在干什么？”教授大叫道。智者说：“我在倒茶。你的思想就像这个茶杯，已经倒满了茶，容不下任何其他东西。你必须先放空你的思想，我们才能继续往下说。”

在我看来，优秀的程序员就像是空的茶杯，他不断地探索着新的技术（noSQL、node-js等等）。普通的程序员沉浸在那些固有的想法中：C++很棒，Java不好，PHP只能用来编写网页，MySQL是数据库的唯一选择。我希望你能够以开放的心态阅读本书，从而发现一些有价值的东西。正如铃木俊隆所说：

在初学者眼中，世界充满了可能；专家眼中，世界大都已经既定。

数据挖掘简介及如何使用本书

想象我们身处一个150年前的美国小镇。大家都互相认识。商店新进了一批布料，店员注意到几块印有特殊花纹的布料肯定会受到克兰西女士的喜爱，因为他知道这位女士喜欢同类型的布料，并暗自记下如果克兰西女士下次到访，要将这块布料推荐给她。温克勒周向酒吧老板威尔逊先生提到，他正考虑要将自己的雷明顿来福枪转售。威尔逊先生将这个消息告诉了巴德巴克莱，他正想购买一把高品质的来福枪。瓦尔克兹警官和他的下属们知道李派是个麻烦人物，因为他总是喝酒，脾气不好又身强力壮。100年前的小镇生活全靠人与人之间的关系。



人们知道你的喜好，健康和婚姻状况。无论好坏，这都是一种个性化的体验。世界各自的社区都存在这种高度个性化的生活状态。

让我们穿越100年，来到1960年代。个性化的交流会有所减少，但它依然存在。一位常客走进书店时，店员会招呼道“米切纳的新书到了”，因为他知道这位常客喜欢米切纳的书。或者他会推荐高华德的《The Conscience of a Conservative》，因为他知道这位常客是位坚定的保守派。再如，来到餐厅的常客会被服务员小姐问道“照旧吗”。

即使在今天，也存在很多个性化的服务。我去附近的梅西亚咖啡店时，服务员会问“您是要一杯加量的大杯拿铁吗”，因为她知道这是我每天早上都会喝的咖啡。当我带着贵宾犬去宠物店修剪毛发时，美容师不需要问我要剪成什么造型，因为他知道我喜欢标准的德国造型。

但时过境迁，如今的生活已和百年前的小镇不一样了。大型购物超市取代了邻家的小型商店或商贩，这使得人们的选择变得有限起来。福特曾说过：“顾客们可以让轿车喷上自己喜欢的颜色，不过前提是他喜欢的是黑色。”音像店只会采购有限的音像制品，书店采购的书也是有限的。想要吃冰激凌？你可以选择香草味、巧克力味，也许草莓味也有。想要买一台洗衣机？1950年的希尔士里只有两种机型：55美元的标准款和95美元的豪华款。

欢迎来到21世纪

到了21世纪，选择范围有限的问题已经不复存在了。想听音乐？iTunes里有1100万首曲目！截止到2011年10月，他们一共售出了160亿首歌曲。还想要有更多选择？可以去Spotify，那里有超过1500万首歌曲。

想买一本书？亚马逊有200万本可供选择。



想看视频？那选择就更多了：Netflix（10万个视频）、Hulu（5万）、Amazon Prime（10万）。







想买一台笔记本电脑？在亚马逊可以搜索到3811条结果。

搜索电饭煲则可以得到1000条结果。

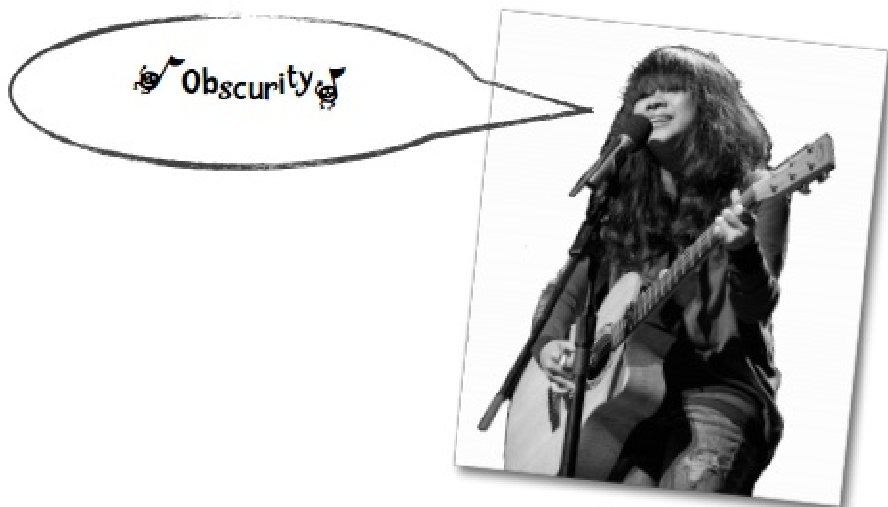


相信在不久的将来会有更多的商品可供选择——上十亿的在线音乐，各种各样的视频节目，以及能够用3D打印机定制的产品。

寻找相关产品

现在的问题在于——如何寻找相关的产品。在那1100万首iTunes曲目中，肯定有一部分音乐是我特别喜爱的，我该如何找到它们？我想在Netflix上观看一段视频，应该看什么呢？我想用P2P下载一部电影，哪部比较好呢？而且问题会越来越严重——每分钟都有数以万记的媒体数据被发布到互联网上；共享群组里每分钟都会新增100个文件；YouTube上每分钟都会有24个小时时长的新视频被上传；每小时会有180本新书发布。每天都有新的东西可以购买，要想找到自己感兴趣的产品变得越来越难。

如果你是一位音乐人——比如马来西亚的季小薇——真正的威胁并不来自于你的专辑被他人非法下载，而是大众根本找不到你的专辑。



那要如何寻找商品呢？

很久以前，在那个小镇里，朋友 会帮助我们寻找商品——那块布料很适合我；那本新书我很喜欢；那台迷你留声机很棒。即便在今天，我们也非常看重朋友的推荐。

我们还会请 专家 帮助我们寻找商品。过去，消费者周刊能够对所有的洗衣机（20种）和电饭煲（10种）做出评测，并进行推荐；但如今，亚马逊上有上百种电饭煲，不是一个专家就能评测完全的。过去，影评家艾伯特几乎能够对所有的电影进行评判；但如今，每年都有两万五千部电影在世界各地上映。此外，我们还能通过各种途径获取到视频节目。艾伯特也好，其他影评家也罢，是不可能对所有的电影做出评价的。

此外，我们还会通过 商品本身 来寻找。比如，我有一台用了三十年的希尔士洗衣机，所以我会再去购买一台同品牌的洗衣机；我喜欢披头士的一张专辑，所以会认为他们的另一张专辑也很有吸引力。

这些寻找商品的方式可以沿用至今，但是我们需要用电算化的手段让这些方法能够适用于**21**世纪的商品数量。 本书将会探索这些方法，将人们的喜恶收集起来，分析他们的购买历史，发掘社会网络（朋友）的数据价值，从而帮助我们找到相关的商品。比方说，我喜欢Phoenix乐队，那系统会使用这个乐队的一些特点——重金属、朋克、和声——来推荐其他的乐队给我，如The Strokes乐队。

不仅仅是寻找商品

数据挖掘不仅仅是用来推荐商品，或是单单给商人增加销量的。看看下面的示例。

回到一百年前的那个小镇，镇长在竞选演讲上可以针对每个选民来给出承诺：玛莎，我知道你对教育事业非常在意，我会尽一切努力去招募另一名教师到我们小镇来；约翰，你的面包房经营得如何？我会在你的商店周围建造更多的停车场的。



我父亲是联合汽车工会的成员。在选举期间，工会的代表曾来到我家，游说我父亲要投票给谁：

> 赛尔，你好。你的家人和孩子都好吧？……现在让我来告诉你为什么要投票给赛德勒，让这位社会学家当选市长。



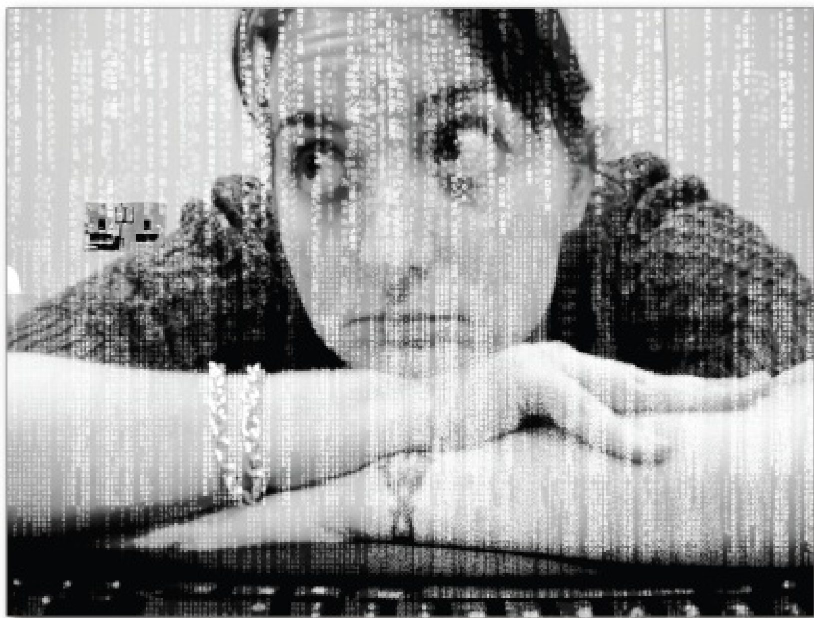
赛德勒是1968至1960年密尔沃基市的市长。

随着电视的普及，这类个性化的推广信息逐渐转变为广告形式，每个人得到的信息都是一样的，其中一个著名的示例是为支持约翰逊竞选的黛西广告（一个小女孩在雏菊花田里骑着单车，此时一枚核弹从天而降）。现在，随着得票率相差得越来越小，以及数据挖掘技术的应用推广，个性化的竞选广告又回来了。比如你对女权主义很在意，也许就会接听到一个关于这方面信息的语音电话。



那个小镇的警官非常清楚谁是制造麻烦的人。而如今，各类威胁是隐秘起来的，恐怖主义随处可能发生。2001年10月11日，政府通过了《美国爱国者法案》（USA Patriot Act，意为提供合适的工具来截获恐怖主义的相关信息，从而保护美国公民）。这项法案的条款之一是调查者能够通过各种渠道来获得信息，比如图书馆借阅记录、旅馆出入记录、信用卡信息、公路收费站记录等等。美国政府通过和某些私营企业合作，收集我们的各项信息。比如赛新公司持有几乎所有人的记录，我们的照片、住址、座驾、收入、消费习惯、朋友等。赛新拥有的超级计算机系统能够通过数据挖掘来预测人们的行为。他们的产品有一个响亮的名字：

矩阵

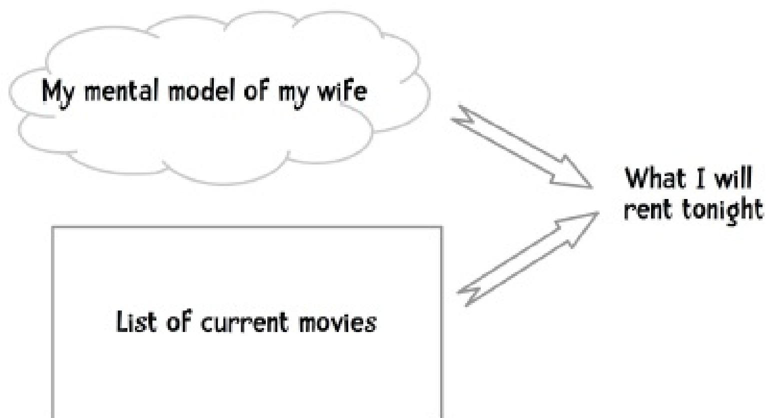


数据挖掘扩展了我们的能力

贝克在他的作品《数学奇才》中写道：

> 想象你正在一家咖啡馆，可能十分嘈杂。一位年轻的女士坐在你的右侧，正在操作笔记本电脑。你转过头去，看着她的屏幕。她正在上网。你开始观察。
>
> 几个小时过去了，她先是阅读了一篇在线论文，然后读了三篇关于中国的文章；她浏览了周五晚上会上映的电影，还看了一篇功夫熊猫的影评；她点击了一个广告，说是能帮助用户找到自己的老同学。你在那里看着她操作，并记录下来。每过一分钟，你对她的了解就多一分。好，现在想象一下你可以同时看着1500万人的电脑屏幕，记录他们的操作。

数据挖掘的重点在于找到数据中的模式。对于少量的数据，我们非常擅长在大脑中构建模型，搜寻模式。比如，今晚我想和妻子看一部电影，我很清楚她喜欢什么类型的电影。我知道她不喜欢含有暴力元素的电影（这就是她不喜欢第九区的原因），她喜欢卡夫曼的电影。我可以利用这些信息来预测她会对什么电影感兴趣。



一位欧洲的朋友远道而来，我知道她是一位素食主义者，所以我能猜到她一定不会喜欢我们当地的烤肋排。人们非常善于利用已有信息来进行预测。数据挖掘则扩展了我们的能力，让我们能够处理海量的数据，比如我上文提到的1500万人的示例。数据挖掘能让潘多拉音乐站提供个性化的音乐列表；它能让Netflix将你最感兴趣的视频推荐给你。

海量数据挖掘不是星际争霸II才有的东西

20世纪末，百万单词的数据已经是很大的量了。我于1990年代毕业（没错，我已经很老了），有一年我作为程序员在研究新约圣经，虽然只有20万字，但仍无法完整地放入主机内存，所以只能将计算结果不断地写入磁带中，而磁带的装卸是需要经过批准的。



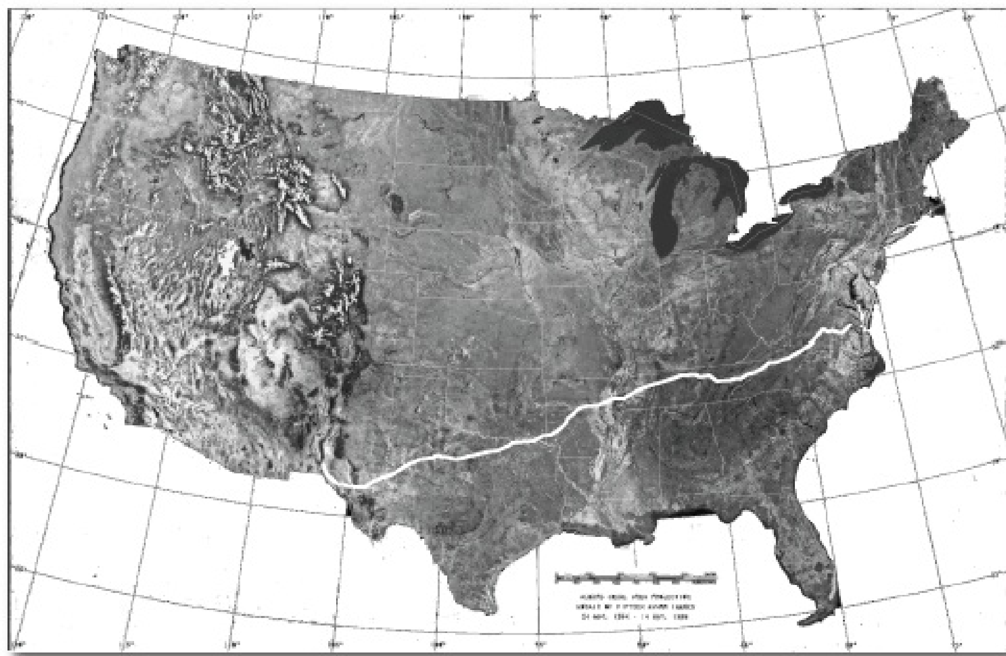
这次的研究成果汇集成了一本书，名为《Analytical Greek New Testament》，由T.福利伯格和B.福利伯格编写。我是当时的三名程序员之一，在明尼苏达大学完成的研究。

如今，在TB级别的数据量上做挖掘已经很常见了。谷歌有超过5PB的页面数据（即5000TB）。2006年，谷歌向研究者社区开放了一万亿单词量的数据集。美国国家安全局有着上万亿的电话录音数据。Acxiom，这家做数据采集的公司（信用卡消费记录、电话通信记录、医疗记录、车辆登记等），有着全美两亿成年人的信息，共计超过1PB的数据。



图为包含了1PB数据的服务器集装箱。

《无处可藏》的作者欧哈罗曾试图帮助我们理解1PB的数据是什么样的概念，说这些数据相当于5万公里的钦定版圣经的长度。我经常往返于新墨西哥州和弗吉尼亚州，两地相距两万公里，于是我便可以想象一路上看到的全是这些书籍，数据量可见之大。



美国国会图书馆有大学20TB的文字，你可以将这些文字全部放入仅需几千美金的硬盘中。相对地，沃尔玛则有超过570TB的数据。这些数据不只是存放在那儿，而是不断有人对其进行挖掘，找到新的关联、新的模式。这就是海量数据挖掘！

本书中我们只会处理很小量的数据，这是好事，因为我们不希望自己的代码运行了一整周后发现其中有一个逻辑错误。我们会处理的最大数据量也在百兆以下，最小的数据集则只有几十行。

本书的结构

这本书按照边学边做的原则编写。与其被动地接受书中的内容，我建议读者使用书中提供的Python代码来进行实践。尝试各种算法，做一些修改，使用不同的数据集查看效果，从而真正地掌握这些知识和技术。



我会尝试在简单易懂的Python代码和其背后的算法逻辑之间找到平衡点。为了避免读者们为各种理论、数学公式、以及Python代码绞尽脑汁，我会增加图表和插画来做调剂。

谷歌研究院总监诺维格曾在他的Udacity课程《计算机程序设计》中写道：

> 我会向你展示和讨论我的解决方案。但需要注意的是，解决问题的方案不止一个。并不是说我的方案是 唯一的 或 最好的 。我的方案不过是帮助你学习编程的一种风格和技术。如果你用另一种方式解决了问题，那会非常好。
>
> 所有的学习过程都是在你的头脑中发生的，而不是我的。所以你需要非常了解你的代码和我的代码之间的关系——你需要自己编写出答案，然后从我的代码中挑选出有用的部分来学习和借鉴。

我非常赞同这个观点



图文：用血和汗水来编程！

这本书并不是一本完整论述数据挖掘技术的教科书。市面上有一些这样的教科书，如由谭恩、 斯坦巴克、以及库马合著的《数据挖掘导论》，就很全面地讲解了数据挖掘的各种理论，及其背后的数学知识。而你正在阅读的这本书，只是帮助你快速了解数据挖掘的基础理论，并进行实践。读完本书后，你可以再找一本完整的教科书来填补空白。

这本书另一个比较实用的地方是它所提供的Python代码和数据集。我认为这可以帮助读者更快速地掌握数据挖掘的核心思想，而又不会陷得太深，事倍功半。

读完本书后你将能够做些什么事？

读完本书后，你将有能力使用Python或其它编程语言，为一个网站设计和实现一套推荐系统。例如，你在亚马逊上浏览一件商品，或者在潘多拉上聆听一首音乐，你可得到一组相关产品的列表（也就是“猜你喜欢”）。你会学到如何开发出这样一套系统。此外，这部分书提到的相关术语可以让你能够顺畅地与数据挖掘团队作沟通。

作为目标的一部分，本书还将为你揭开推荐系统的神秘面纱，包括那些恐怖分子识别系统及其他数据挖掘系统，至少你将知道这些系统是怎么运作的。

为什么这点很重要？

为什么你需要花时间来阅读本书呢？在本章的开始，我给出了很多示例来说明数据挖掘的重要性。那段文字可以转述如下：市场上有很多商品（电影、音乐、书籍、烹饪器具），而且数量在不断增加，随之而来的问题便是如何在这么多商品中找到我们最感兴趣的——那么多电影我该看哪部？我接下来应该读哪本书？数据挖掘就是用来解决这类问题的。大多数网站

都会提供查找商品的功能，除了上面提到的商品，你还会考虑该关注哪位好友；是否能够有一份报纸只刊登你感兴趣的文章？如果你是一名Web开发者，就非常需要了解数据挖掘方面的知识了。

好，现在你应该了解为什么要花时间来学习数据挖掘了，但为何要选择这本书呢？市面上有些书籍是非技术类的，描述了数据挖掘的大致情况。这些书可以快速翻阅，十分有趣，而且不贵，很适合深夜阅读（因为没有繁杂的技术细节）。这类书籍的最佳代表是贝克的《数学奇才》，我非常推荐这本书。在我往来弗吉尼亚和新墨西哥时就听的是这本书的语音版。另一个极端则是数据挖掘教学中使用的教科书。这些书籍涵盖面广，将数据挖掘的理论和实践讲解得非常透彻，所以我也推荐阅读这类书籍。至于这本书，则是用来填补这两者之间的空白的。本书的目标读者是那些喜欢编程的骇客们。



这本书应该在电脑前阅读，这样读者就可以立刻编写代码参与其中。

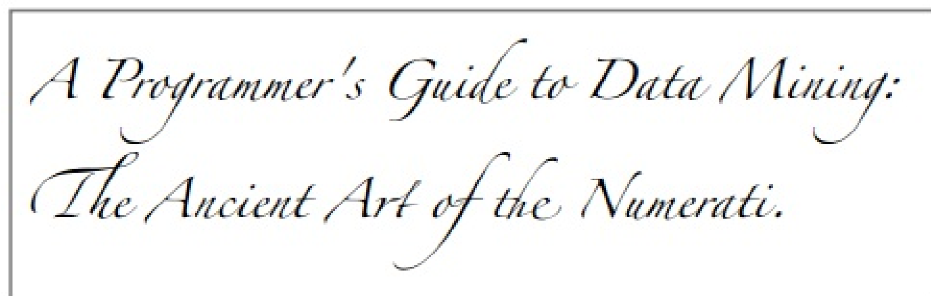
$$s(i,j)=\frac{\sum_{u\in U}(R_{u,i}-\overline{R}_u)(R_{u,j}-\overline{R}_u)}{\sqrt{\sum_{u\in U}(R_{u,i}-\overline{R}_u)^2}\sqrt{\sum_{u\in U}(R_{u,j}-\overline{R}_u)^2}}$$

天呐，这是什么？这本书会包含一些数学公式，不过我会用一种简明的方式表述，相信普通的程序员都能了解，即便你已经忘记了大学中学习数学知识。

如果以上这些都不能说服你，那还有一点：这本书是免费的，你可以随意分享它。

标题中的“数学奇才的古老艺术”有什么含义？

2010年6月，我曾尝试给这本书起一个合适的标题。我喜欢有趣的标题，但很可惜我不太擅长起名字。近期，我发表了篇关于数据挖掘的论文，名为《扎入文字堆：阿拉伯文字的地域化分类》。我喜欢这个标题，不过我得承认这是我的太太帮我取的。我曾和马克肖恩合著了一篇论文，名为《情绪与模式：从理论到争辩》，这个标题也是我的搭档取的。总之，六月时我取的那些标题很难一眼看出这本书讲的是什么，所以我最后用了《面向程序员的数据挖掘指南》作为标题的一部分，因为这个标题和本书的内容非常契合——这本书是提供给正在从事编程工作的人员阅读的。也许你会疑惑子标题究竟是什么意思：



数学奇才（Numerati）是贝克自己创造的一个词语。如今，我们每个人无时无刻不在创造着新的数据，信用卡购物记录、推特、格瓦拉上的博客、Foursquare上的签到、手机通话记录、电子邮件、文字短信等。

当你一早醒来，“矩阵”就知道你会乘坐雾谷站7:10的地铁，并于7:32在西站下车；矩阵知道7:45分你会去第五大街的星巴克买上一杯大杯拿铁和一份蓝莓饼；8:05分，你用格瓦拉在上班地点签到；9:35分，你在亚马逊上购买了一套瘦身教程DVD和一副门上单杠；你在Golden Falafel吃的午餐。

贝克在书中这样写道：

> 只有那些数学家、计算机科学家、以及工程师们才能从这些庞大的数据集中获得有用的信息。这些数学奇才会从这些数据中了解到什么？首先，他们能够准确地定位到我们。比如你是纽约北部市郊的一个潜在的SUV客户，或是一个经常去教堂做礼拜的人，或是阿尔伯克基市的一名反堕胎的民主党人士；也许你是一个即将被调任到海得拉巴市的一名Java工程师，或是一个热爱爵士乐的人；你是射手座的，喜欢喝勤地酒，想在乡野间漫步，最后在斯德哥尔摩的篝火旁酣睡；更夸张的，也许你腰绑炸弹，正乘上一部公交车。无论你是谁，处在茫茫人海中，那些公司或政府机构都能掌握你的行踪。

你可能猜到了，起这个标题是因为我喜欢贝克的这段描述。



第二章：推荐系统入门

原文：http://guidetodatamining.com/chapter-2/

内容：

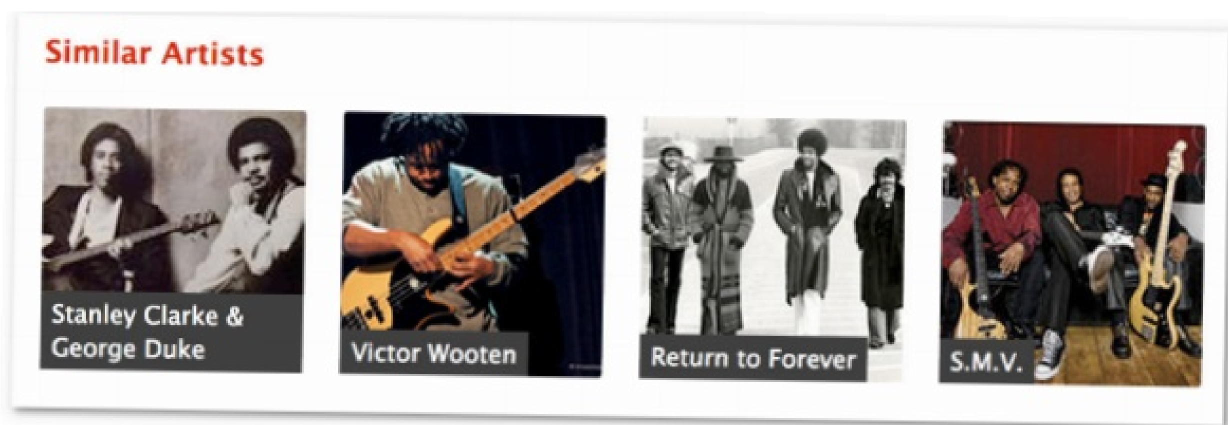
- 推荐系统工作原理
- 社会化协同过滤工作原理
- 如何找到相似物品
- 曼哈顿距离
- 欧几里得距离
- 闵可夫斯基距离
- 皮尔逊相关系数
- 余弦相似度
- 使用Python实现K最邻近算法
- 图书漂流站（BookCrossing）数据集

你喜欢的东西我也喜欢

我们将从推荐系统开始，开启数据挖掘之旅。推荐系统无处不在，如亚马逊网站的“看过这件商品的顾客还购买过”板块：



last.fm上对音乐和演唱会的推荐（相似歌手）：



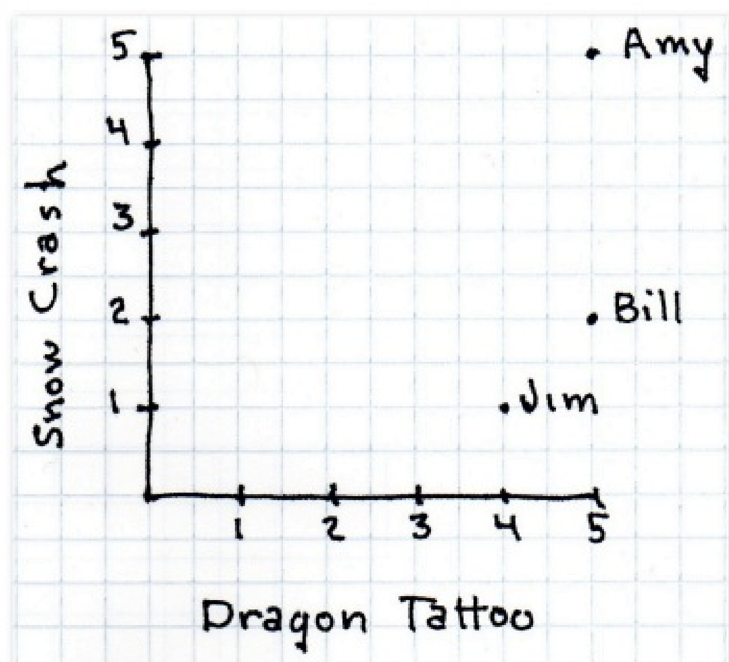


在亚马逊的例子里，它用了两个元素来进行推荐：一是我浏览了里维斯翻译的《法华经》一书；二是其他浏览过该书的顾客还浏览过的译作。

本章我们讲述的推荐方法称为协同过滤。顾名思义，这个方法是利用他人的喜好来进行推荐，也就是说，是大家一起产生的推荐。他的工作原理是这样的：如果要推荐一本书给你，我会在网站上查找一个和你类似的用户，然后将他喜欢的书籍推荐给你——比如巴奇加卢比的《发条女孩》。

如何找到相似的用户？

所以首先要做的工作是找到相似的用户。这里用最简单的二维模型来描述。假设用户会在网站用五颗星来评价一本书——没有星表示书写得很糟，五颗星表示很好。因为我们用的是二维模型，所以仅对两本书进行评价：史蒂芬森的《雪崩》（纵轴）和拉尔森的《龙纹身的女孩》（横轴）。



首先，下表显示有三位用户对这两本书做了评价：

	Snow Crash	Girl with the Dragon Tattoo
Amy	5☆	5☆
Bill	2☆	5☆
Jim	1☆	4☆

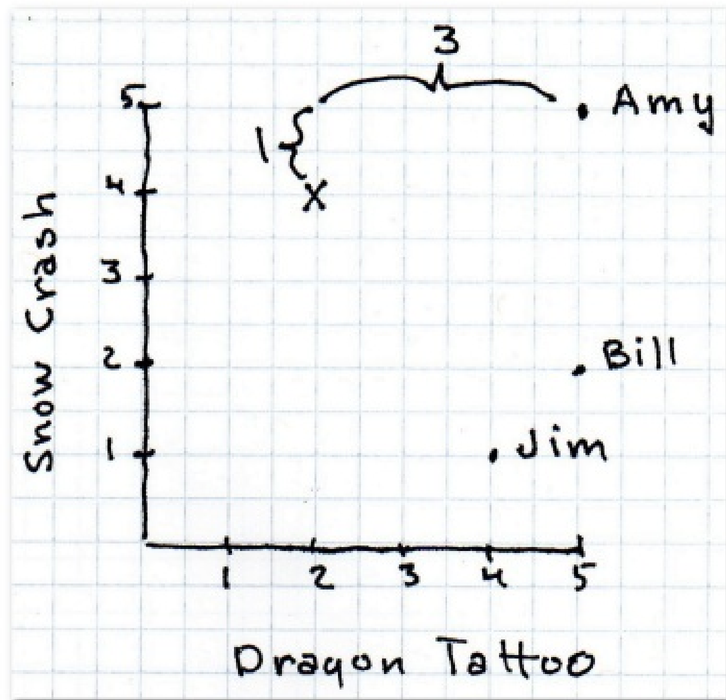
现在我想为神秘的X先生推荐一本书，他给《雪崩》打了四星，《龙纹身的女孩》两星。第一个任务是找出哪个用户和他最为相似。我们用距离来表示。

曼哈顿距离

最简单的距离计算方式是曼哈顿距离。在二维模型中，每个人都可以用(x, y)的点来表示，这里我用下标来表示不同的人，(x_1, y_1)表示艾米，(x_2, y_2)表示那位神秘的X先生，那么他们之间的曼哈顿距离就是：

$$|x_1 - x_2| + |y_1 - y_2|$$

也就是x之差的绝对值加上y之差的绝对值，这样他们的距离就是4。



完整的计算结果如下：

	Distance From Ms. X
Amy	4
Bill	5
Jim	5

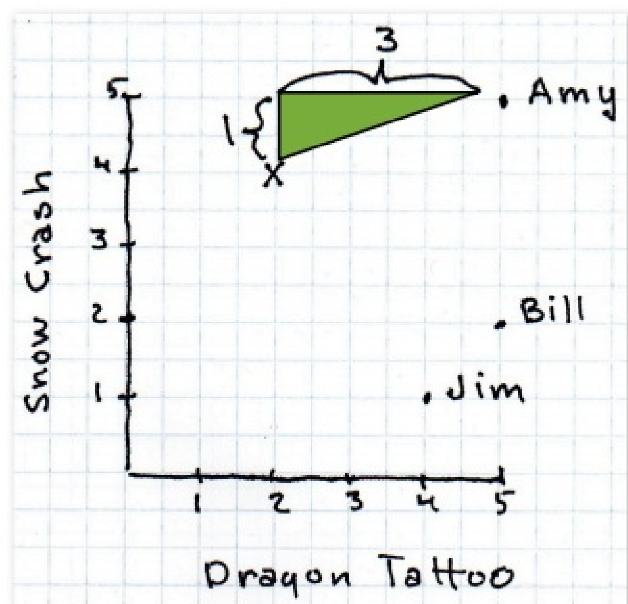
艾米的距离最近，在她的浏览历史中可以看到她曾给巴奇加卢比的《发条女孩》打过五星，于是我们就可以把这本书推荐给X先生。

欧几里得距离

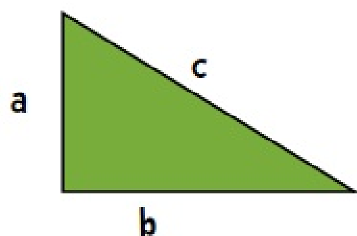
曼哈顿距离的优点之一是计算速度快，对于Facebook这样需要计算百万用户之间的相似度时就非常有利。

勾股定理

也许你还隐约记得勾股定理。另一种计算距离的方式就是看两点之间的直线距离：



利用勾股定理，我们可以如下计算距离：



$$c = \sqrt{a^2 + b^2}$$

这条斜线就是欧几里得距离，公式是：

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

回顾一下，这里的x_1表示用户1喜欢《龙纹身》的程度，x_2是用户2喜欢这本书的程度；y_1则是用户1喜欢《雪崩》的程度，y_2是用户2喜欢这本书的程度。

艾米给《龙纹身》和《雪崩》都打了五颗星，神秘的X先生分别打了两星和四星，这样他们之间的欧几里得距离就是：

$$\sqrt{(5-2)^2 + (5-4)^2} = \sqrt{3^2 + 1^2} = \sqrt{10} = 3.16$$

以下是全部用户的计算结果：

	Distance from Ms. X
Amy	3.16
Bill	3.61
Jim	3.61

N维模型

刚才我们仅仅对两本书进行评价（二维模型），下面让我们扩展一下，尝试更复杂的模型。假设我们现在要为一个在线音乐网站的用户推荐乐队。用户可以用1至5星来评价一个乐队，其中包含半星（如2.5星）。下表展示了8位用户对8支乐队的评价：

	Angelica	Bill	Chan	Dan	Hailey	Jordyn	Sam	Veronica
Blues Traveler	3.5	2	5	3	-	-	5	3
Broken Bells	2	3.5	1	4	4	4.5	2	-
Deadmau5	-	4	1	4.5	1	4	-	-
Norah Jones	4.5	-	3	-	4	5	3	5
Phoenix	5	2	5	3	-	5	5	4
Slightly Stoopid	1.5	3.5	1	4.5	-	4.5	4	2.5
The Strokes	2.5	-	-	4	4	4	5	3
Vampire Weekend	2	3	-	2	1	4	-	-

表中的短横表示这位用户没有给这支乐队打分。我们在计算两个用户的距离时，只采用他们都评价过的乐队，比如要计算Angelica和Bill的距离，我们只会用到5支乐队。这两个用户的曼哈顿距离为：

	Angelica	Bill	Difference
Blues Traveler	3.5	2	1.5
Broken Bells	2	3.5	1.5
Deadmau5	-	4	
Norah Jones	4.5	-	
Phoenix	5	2	3
Slightly Stoopid	1.5	3.5	2
The Strokes	2.5	-	-
Vampire Weekend	2	3	1
Manhattan Distance:			9

最后距离即是上方数据的加和：(1.5 + 1.5 + 3 + 2 + 1)。

计算欧几里得距离的方法也是类似的，我们也只取双方都评价过的乐队。

	Angelica	Bill	Difference	Difference²
Blues Traveler	3.5	2	1.5	2.25
Broken Bells	2	3.5	1.5	2.25
Deadmau5	-	4		
Norah Jones	4.5	-		
Phoenix	5	2	3	9
Slightly Stoopid	1.5	3.5	2	4
The Strokes	2.5	-	-	
Vampire Weekend	2	3	1	1
Sum of squares				18.5
Euclidean Distance				4.3

用公式来描述即：

$$Euclidean = \sqrt{(3.5-2)^2 + (2-3.5)^2 + (5-2)^2 + (1.5-3.5)^2 + (2-3)^2}$$

$$= \sqrt{1.5^2 + (-1.5)^2 + 3^2 + (-2)^2 + (-1)^2}$$

$$= \sqrt{2.25 + 2.25 + 9 + 4 + 1}$$

$$= \sqrt{18.5} = 4.3$$

掌握了吗？那就试试计算其他几个用户之间的距离吧。



有个瑕疵

当我们计算Hailey和Veronica的距离时会发现一个问题：他们共同评价的乐队只有两支（Norah Jones和The Strokes），而Hailey和Jordyn共同评价了五支乐队，这似乎会影响我们的计算结果，因为Hailey和Veronica之间是二维的，而Haily和Veronica之间是五维的。曼哈顿距离和欧几里得距离在数据完整的情况下效果最好。如何处理缺失数据，这在研究领域仍是一个活跃的话题。本书的后续内容会进行一些讨论，这里先不展开。现在，让我们开始构建一个推荐系统吧。

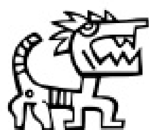
推广：闵可夫斯基距离

我们可以将曼哈顿距离和欧几里得距离归纳成一个公式，这个公式称为闵可夫斯基距离：

$$d(x,y)=(\sum_{k=1}^{n}|x_k-y_k|^r)^{\frac{1}{r}}$$

其中：

- `r = 1` 该公式即曼哈顿距离
- `r = 2` 该公式即欧几里得距离
- `r = ∞` 极大距离



Arghhhh Math!



当你在书中看到这些数学公式，你可以选择快速略过它，继续读下面的文字，过去我就是这样；你也可以停下来，好好分析一下这些公式，会发现其实它们并不难理解。比如上面的公式，当r = 1时，可以简化成如下形式：

$$d(x,y)=\sum_{k=1}^{n}|x_k-y_k|$$

仍用上文的音乐站点为例，x和y分别表示两个用户，d(x, y)表示他们之间的距离，n表示他们共同评价过的乐队数量，我们之前已经做过计算：

	Angelica	Bill	Difference
Blues Traveler	3.5	2	1.5
Broken Bells	2	3.5	1.5
Deadmau5	-	4	
Norah Jones	4.5	-	
Phoenix	5	2	3
Slightly Stoopid	1.5	3.5	2
The Strokes	2.5	-	-
Vampire Weekend	2	3	1
Manhattan Distance:			9

其中Difference一栏表示两者评分之差的绝对值，加起来等于9，也就是他们之间的距离。

当r = 2时，我们得到欧几里得距离的计算公式：

$$d(x,y)=\sqrt{\sum_{k=1}^{n}(x_k-y_k)^2}$$

提前预告一下：r值越大，单个维度的差值大小会对整体距离有更大的影响。



使用Python代码来表示数据（终于要开始编程了）

在Python中，我们可以用多种方式来描述上表中的数据，这里我选择Python的字典类型（或者称为关联数组、哈希表）。

注：本书的所有代码可以在这里找到。

```
users = {"Angelica": {"Blues Traveler": 3.5, "Broken Bells": 2.0, "Norah Jones": 4.5, "Ph
         "Bill":{"Blues Traveler": 2.0, "Broken Bells": 3.5, "Deadmau5": 4.0, "Phoenix":
         "Chan": {"Blues Traveler": 5.0, "Broken Bells": 1.0, "Deadmau5": 1.0, "Norah Jon
         "Dan": {"Blues Traveler": 3.0, "Broken Bells": 4.0, "Deadmau5": 4.5, "Phoenix":
         "Hailey": {"Broken Bells": 4.0, "Deadmau5": 1.0, "Norah Jones": 4.0, "The Stroke
         "Jordyn":  {"Broken Bells": 4.5, "Deadmau5": 4.0, "Norah Jones": 5.0, "Phoenix":
         "Sam": {"Blues Traveler": 5.0, "Broken Bells": 2.0, "Norah Jones": 3.0, "Phoenix
         "Veronica": {"Blues Traveler": 3.0, "Norah Jones": 5.0, "Phoenix": 4.0, "Slightl
        }
```

我们可以用以下方式来获取某个用户的评分：

```
>>> users["Veronica"]
{"Blues Traveler": 3.0, "Norah Jones": 5.0, "Phoenix": 4.0, "Slightly Stoopid": 2.5, "The
>>>
```

计算曼哈顿距离

```
def manhattan(rating1, rating2):
    """计算曼哈顿距离。rating1和rating2参数中存储的数据格式均为
    {'The Strokes': 3.0, 'Slightly Stoopid': 2.5}"""
    distance = 0
    for key in rating1:
        if key in rating2:
            distance += abs(rating1[key] - rating2[key])
    return distance
```

我们可以做一下测试：

```
>>> manhattan(users['Hailey'], users['Veronica'])
2.0
>>> manhattan(users['Hailey'], users['Jordyn'])
7.5
>>>
```

下面我们编写一个函数来找出距离最近的用户（其实该函数会返回一个用户列表，按距离排序）：

```
def computeNearestNeighbor(username, users):
    """计算所有用户至username用户的距离，倒序排列并返回结果列表"""
    distances = []
    for user in users:
        if user != username:
            distance = manhattan(users[user], users[username])
            distances.append((distance, user))
    # 按距离排序——距离近的排在前面
    distances.sort()
    return distances
```

简单测试一下：

```
>>> computeNearestNeighbor("Hailey", users)
[(2.0, 'Veronica'), (4.0, 'Chan'), (4.0, 'Sam'), (4.5, 'Dan'), (5.0, 'Angelica'), (5.5, '
```

最后，我们结合以上内容来进行推荐。假设我想为Hailey做推荐，这里我找到了离他距离最近的用户Veronica。然后，我会找到出Veronica评价过但Hailey没有评价的乐队，并假设Hailey对这些陌生乐队的评价会和Veronica相近。比如，Hailey没有评价过Phoenix乐队，而Veronica对这个乐队打出了4分，所以我们认为Hailey也会喜欢这支乐队。下面的函数就实现了这一逻辑：

```
def recommend(username, users):
    """返回推荐结果列表"""
    # 找到距离最近的用户
    nearest = computeNearestNeighbor(username, users)[0][1]
    recommendations = []
    # 找出这位用户评价过、但自己未曾评价的乐队
    neighborRatings = users[nearest]
    userRatings = users[username]
    for artist in neighborRatings:
        if not artist in userRatings:
            recommendations.append((artist, neighborRatings[artist]))
    # 按照评分进行排序
    return sorted(recommendations, key=lambda artistTuple: artistTuple[1], reverse = True
```

下面我们就可以用它来为Hailey做推荐了：

```
>>> recommend('Hailey', users)
[('Phoenix', 4.0), ('Blues Traveler', 3.0), ('Slightly Stoopid', 2.5)]
```

运行结果和我们的预期相符。我们看可以看到，和Hailey距离最近的用户是Veronica，Veronica对Phoenix乐队打了4分。我们再试试其他人：

```
>>> recommend('Chan', users)
[('The Strokes', 4.0), ('Vampire Weekend', 1.0)]
>>> recommend('Sam', users)
[('Deadmau5', 1.0)]
```

我们可以猜想Chan会喜欢The Strokes乐队，而Sam不会太欣赏Deadmau5。

```
>>> recommend('Angelica', users)
[]
```

对于Angelica，我们得到了空的返回值，也就是说我们无法对其进行推荐。让我们看看是哪里有问题：

```
>>> computeNearestNeighbor('Angelica', users)
[(3.5, 'Veronica'), (4.5, 'Chan'), (5.0, 'Hailey'), (8.0, 'Sam'), (9.0, 'Bill'), (9.0, 'D
```

Angelica最相似的用户是Veronica，让我们回头看看数据：

	Angelica	Bill	Chan	Dan	Hailey	Jordyn	Sam	Veronica
Blues Traveler	3.5	2	5	3	-	-	5	3
Broken Bells	2	3.5	1	4	4	4.5	2	-
Deadmau5	-	4	1	4.5	1	4	-	-
Norah Jones	4.5	-	3	-	4	5	3	5
Phoenix	5	2	5	3	-	5	5	4
Slightly Stoopid	1.5	3.5	1	4.5	-	4.5	4	2.5
The Strokes	2.5	-	-	4	4	4	5	3
Vampire Weekend	2	3	-	2	1	4	-	-

我们可以看到，Veronica评价过的乐队，Angelica也都评价过了，所以我们没有推荐。

之后，我们会讨论如何解决这一问题。

作业：实现一个计算闵可夫斯基距离的函数，并在计算用户距离时使用它。

```python
def minkowski(rating1, rating2, r):
    distance = 0
    for key in rating1:
        if key in rating2:
            distance += pow(abs(rating1[key] - rating2[key]), r)
    return pow(distance, 1.0 / r)

# 修改computeNearestNeighbor函数中的一行
distance = minkowski(users[user], users[username], 2)
# 这里2表示使用欧几里得距离
```

用户的问题

让我们仔细看看用户对乐队的评分，可以发现每个用户的打分标准非常不同：

- Bill没有打出极端的分数，都在2至4分之间；
- Jordyn似乎喜欢所有的乐队，打分都在4至5之间；
- Hailey是一个有趣的人，他的分数不是1就是4。

那么，如何比较这些用户呢？比如Hailey的4分相当于Jordan的4分还是5分呢？我觉得更接近5分。这样一来就会影响到推荐系统的准确性了。



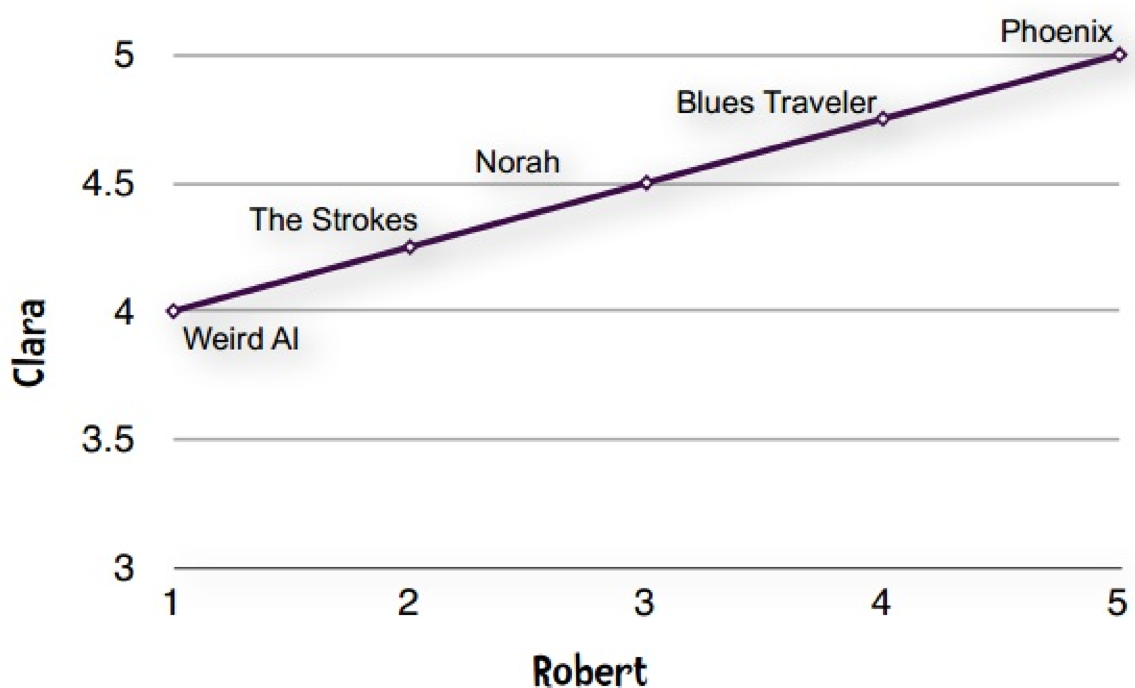
- 左：我非常喜欢Broken Bells乐队，所以我给他们打4分！
- 右：Broken Bells乐队还可以，我打4分。

皮尔逊相关系数

解决方法之一是使用皮尔逊相关系数。简单起见，我们先看下面的数据（和之前的数据不同）：

	Blues Traveler	Norah Jones	Phoenix	The Strokes	Weird Al
Clara	4.75	4.5	5	4.25	4
Robert	4	3	5	2	1

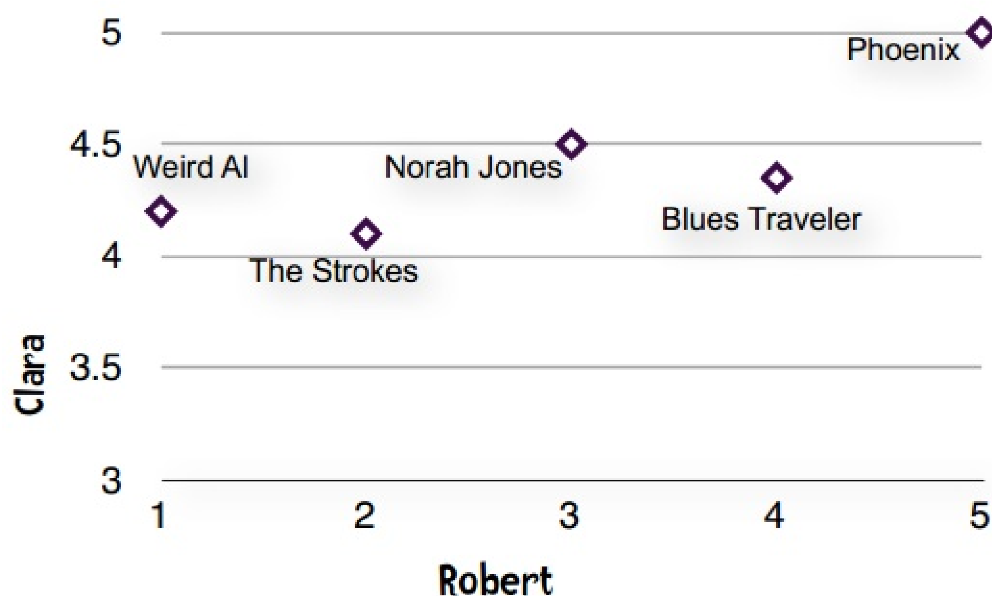
这种现象在数据挖掘领域称为“分数膨胀”。Clara最低给了4分——她所有的打分都在4至5分之间。我们将它绘制成图表：



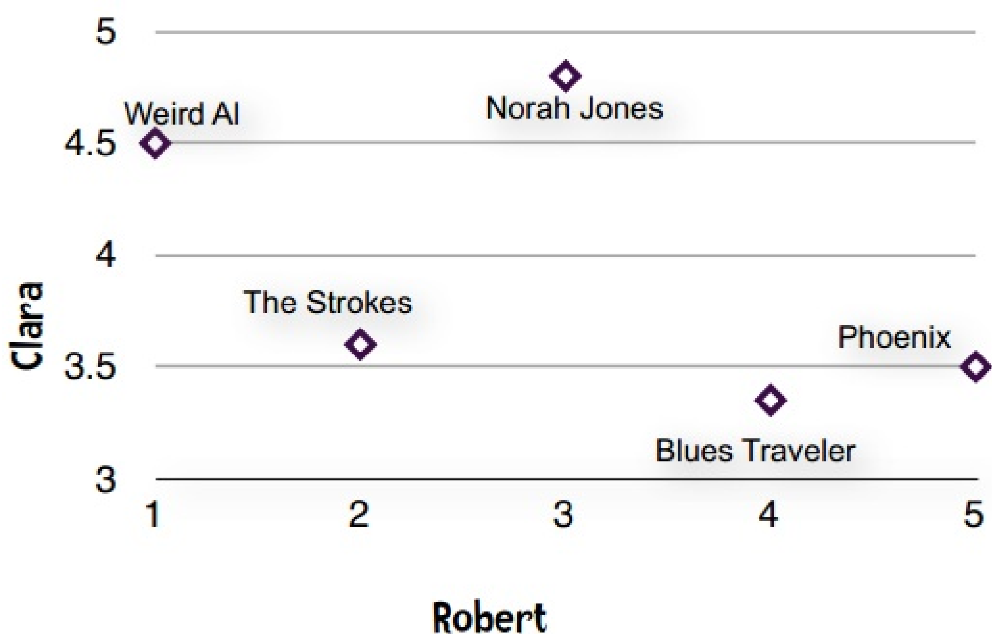
一条直线——完全吻合！！！

直线即表示Clara和Robert的偏好完全一致。他们都认为Phoenix是最好的乐队，然后是Blues Traveler、Norah Jones。如果Clara和Robert的意见不一致，那么落在直线上的点就越少。

意见基本一致的情形



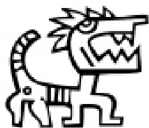
意见不太一致的情形



所以从图表上理解，意见相一致表现为一条直线。皮尔逊相关系数用于衡量两个变量之间的相关性（这里的两个变量指的是Clara和Robert），它的值在-1至1之间，1表示完全吻合，-1表示完全相悖。从直观上理解，最开始的那条直线皮尔逊相关系数为1，第二张是0.91，第三张是0.81。因此我们利用这一点来找到相似的用户。

皮尔逊相关系数的计算公式是：

$$r=\frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}}$$



Arghhhh Math Again!



这里我说说自己的经历。我大学读的是现代音乐艺术，课程包括芭蕾、现代舞、服装设计等，没有任何数学课程。我高中读的是男子学校，学习了管道工程和汽车维修，只懂得很基础的数学知识。不知是因为我的学科背景，还是习惯于用直觉来思考，当我遇到这样的数学公式时会习惯性地跳过，继续读下面的文字。如果你和我一样，我强烈建议你与这种惰性抗争，试着去理解这些公式。它们虽然看起来很复杂，但还是能够被常人所理解的。

上面的公式除了看起来比较复杂，另一个问题是要获得计算结果必须对数据做多次遍历。好在我们有另外一个公式，能够计算皮尔逊相关系数的近似值：

$$r=\frac{\sum_{i=1}^{n}x_iy_i-\frac{\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i}{n}}{\sqrt{\sum_{i=1}^{n}x_i^2-\frac{(\sum_{i=1}^{n}x_i)^2}{n}}\sqrt{\sum_{i=1}^{n}y_i^2-\frac{(\sum_{i=1}^{n}y_i)^2}{n}}}$$

这个公式虽然看起来更加复杂，而且其计算结果会不太稳定，有一定误差存在，但它最大的优点是，用代码实现的时候可以只遍历一次数据，我们会在下文看到。首先，我们将这个公式做一个分解，计算下面这个表达式的值：

$$\sum_{i=1}^{n}x_iy_i$$

对于Clara和Robert，我们可以得到：

$$(4.75\times4)+(4.5\times3)+(5\times5)+(4.25\times2)+(4\times1)$$

$$=19+13.5+25+8.5+4=70$$

很简单把？下面我们计算这个公式：

$$\frac{\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i}{n}$$

Clara的总评分是22.5， Robert是15，他们评价了5支乐队，因此：

$$\frac{22.5\times15}{5}=67.5$$

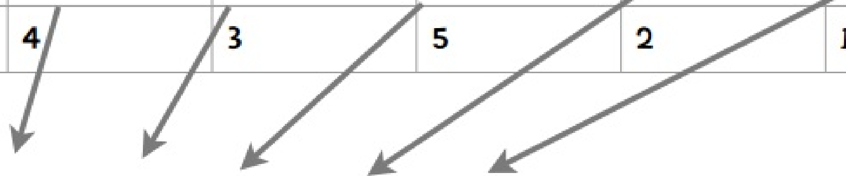
所以，那个巨型公式的分子就是70 - 67.5 = 2.5。

下面我们来看分母：

$$\sqrt{\sum_{i=1}^{n}x_i^2-\frac{(\sum_{i=1}^{n}x_i)^2}{n}}$$

首先：

	Blues Traveler	Norah Jones	Phoenix	The Strokes	Weird Al
Clara	4.75	4.5	5	4.25	4
Robert	4	3	5	2	1



$$\sum_{i=1}^{n}x_i^2=(4.75)^2+(4.5)^2+(5)^2+(4.25)^2+(4)^2=101.875$$

我们已经计算过Clara的总评分是22.5，它的平方是506.25，除以乐队的数量5，得到101.25。综合得到：

$$\sqrt{101.875-101.25}=\sqrt{.625}=.79057$$

对于Robert，我们用同样的方法计算：

$$\sqrt{\sum_{i=1}^{n} y_i^2 - \frac{(\sum_{i=1}^{n} y_i)^2}{n}} = \sqrt{55 - 45} = 3.162277$$

最后得到：

$$r = \frac{2.5}{.79057(3.162277)} = \frac{2.5}{2.5} = 1.00$$

因此，1表示Clara和Robert的偏好完全吻合。

先休息一下吧



计算皮尔逊相关系数的代码

```
from math import sqrt

def pearson(rating1, rating2):
    sum_xy = 0
    sum_x = 0
    sum_y = 0
    sum_x2 = 0
    sum_y2 = 0
    n = 0
    for key in rating1:
        if key in rating2:
            n += 1
            x = rating1[key]
            y = rating2[key]
            sum_xy += x * y
            sum_x += x
            sum_y += y
            sum_x2 += pow(x, 2)
            sum_y2 += pow(y, 2)
    # 计算分母
    denominator = sqrt(sum_x2 - pow(sum_x, 2) / n) * sqrt(sum_y2 - pow(sum_y, 2) / n)
    if denominator == 0:
        return 0
    else:
        return (sum_xy - (sum_x * sum_y) / n) / denominator
```

测试一下：

```
>>> pearson(users['Angelica'], users['Bill'])
-0.9040534990682699
>>> pearson(users['Angelica'], users['Hailey'])
0.42008402520840293
>>> pearson(users['Angelica'], users['Jordyn'])
0.7639748605475432
```

最后一个公式：余弦相似度

这里我将奉上最后一个公式：余弦相似度。它在文本挖掘中应用得较多，在协同过滤中也会使用到。为了演示如何使用该公式，我们换一个示例。这里记录了每个用户播放歌曲的次数，我们用这些数据进行推荐：

	number of plays		
	The Decemberists The King is Dead	Radiohead The King of Limbs	Katy Perry E.T.
Ann	10	5	32
Ben	15	25	1
Sally	12	6	27

简单扫一眼上面的数据（或者用之前讲过的距离计算公式），我们可以发现Ann的偏好和Sally更为相似。

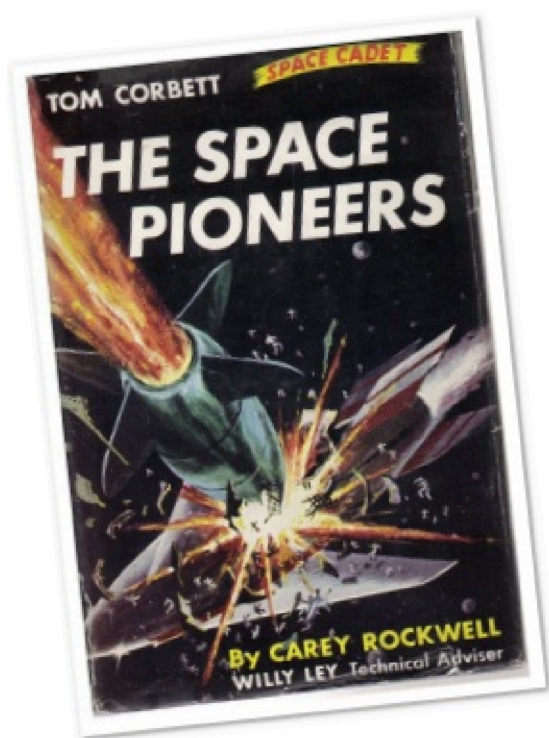
问题在哪儿？

我在iTunes上有大约4000首歌曲，下面是我最常听的音乐：

✓	Name	Time	Artist	Album	Genre	Plays ▾
✓	Moonlight Sonata	7:38	Marcus Miller	Silver Rain	Jazz+Funk	25
✓	Blast!	5:43	Marcus Miller	Marcus	Jazz	20
✓	Art Isn't Real (City of Sin)	2:48	Deer Tick	War Elephant	Alt-Country	19
✓	Between the Lines	4:35	Sara Bareilles	Little Voice	Folk	19
✓	Stay Around A Little Longer (Feat. B.B. King)	5:00	BUDDY GUY	Living Proof	Blues	18
✓	My Companjera	3:22	Gogol Bordello	Trans-Continental...	Alternative...	18
✓	Rebellious Love	3:57	Gogol Bordello	Trans-Continental...	Alternative...	18
✓	Immigraniada (We Comin' Rougher)	3:46	Gogol Bordello	Trans-Continental...	Alternative...	18
✓	Love Song	4:19	Sara Bareilles	Little Voice	Folk	18

可以看到，Moonlight Sonata这首歌我播放了25次，但很有可能你一次都没有听过。事实上，上面列出的这些歌曲可能你一首都没听过。此外，iTunes上有1500万首音乐，而我只听过4000首。所以说单个用户的数据是 稀疏 的，因为非零值较总体要少得多。当我们用1500万

首歌曲来比较两个用户时，很有可能他们之间没有任何交集，这样一来就无从计算他们之间的距离了。



类似的情况是在计算两篇文章的相似度时。比如说我们想找一本和《The Space Pioneers》相类似的书，方法之一是利用单词出现的频率，即统计每个单词在书中出现的次数占全书单词的比例，如“the”出现频率为6.13%，“Tom” 0.89%，“space” 0.25%。我们可以用这些数据来寻找一本相近的书。但是，这里同样有数据的稀疏性问题。《The Space Pioneers》中有6629个不同的单词，但英语语言中有超过100万个单词，这样一来非零值就很稀少了，也就不能计算两本书之间的距离。

余弦相似度的计算中会略过这些非零值。它的计算公式是：

$$\cos(x,y) = \frac{x \cdot y}{||x|| \times ||y||}$$

其中，“·”号表示数量积。“||x||”表示向量x的模，计算公式是：

$$\sqrt{\sum_{i=1}^{n} x_i^2}$$

我们用上文中“偏好完全一致”的示例：

	Blues Traveler	Norah Jones	Phoenix	The Strokes	Weird Al
Clara	4.75	4.5	5	4.25	4
Robert	4	3	5	2	1

所以两个向量为：

$$x=(4.75,4.5,5,4.25,4)$$
$$y=(4,3,5,2,1)$$

它们的模是：

$$\|x\|=\sqrt{4.75^2+4.5^2+5^2+4.25^2+4^2}=\sqrt{101.875}=10.09$$

$$\|y\|=\sqrt{4^2+3^2+5^2+2^2+1^2}=\sqrt{55}=7.416$$

数量积的计算：

$$x\cdot y=(4.75\times 4)+(4.5\times 3)+(5\times 5)+(4.25\times 2)+(4\times 1)=70$$

因此余弦相似度是：

$$\cos(x,y)=\frac{70}{10.093\times 7.416}=\frac{70}{74.85}=0.935$$

余弦相似度的范围从1到-1，1表示完全匹配，-1表示完全相悖。所以0.935表示匹配度很高。

作业：尝试计算**Angelica**和**Veronica**的余弦相似度

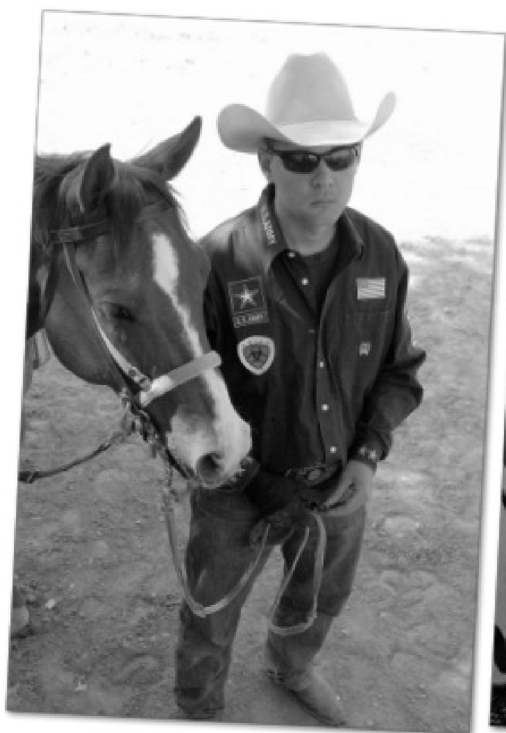
应该使用哪种相似度？

我们整本书都会探索这个问题，以下是一些提示：



- 如果数据存在“分数膨胀”问题，就使用皮尔逊相关系数。
- 如果数据比较“密集”，变量之间基本都存在公有值，且这些距离数据是非常重要的，那就使用欧几里得或曼哈顿距离。
- 如果数据是稀疏的，则使用余弦相似度。

所以，如果数据是密集的，曼哈顿距离和欧几里得距离都是适用的。那么稀疏的数据可以使用吗？我们来看一个也和音乐有关的示例：假设有三个人，每人都给100首音乐评过分。



Jake: hardcore Fan of Country



Linda and Eric: love, love, love 60s rock!

- Jake（左）：乡村音乐的忠实听众。
- Linda和Eric（右）：我们爱六十年代的摇滚乐！

Linda和Eric喜欢相同的音乐，他们的评分列表中有20首相同的的歌曲，且评分均值相差不到0.5！所以他们之间的曼哈顿距离为20 x 0.5 = 10，欧几里得距离则为：

$$d = \sqrt{(.5)^2 \times 20} = \sqrt{.25 \times 20} = \sqrt{5} = 2.236$$

Linda和Jake只共同评分了一首歌曲：Chris Cagle的 *What a Beautiful Day* 。Linda打了3分，Jake打了5分，所以他们之间的曼哈顿距离为2，欧几里得距离为：

$$d = \sqrt{(3-5)^2} = \sqrt{4} = 2$$

所以不管是曼哈顿距离还是欧几里得距离，Jake都要比Eric离Linda近，这不符合实际情况。



> 嘿，我想到一个办法。人们给音乐打分是从1到5分，那些没有打分的音乐就统一给0分好了，这样就能解决数据稀疏的问题了！

想法不错，但是这样做也不行。为了解释这一问题，我们再引入两个人到例子里来：Cooper和Kelsey。他们和Jake都有着非常相似的音乐偏好，其中Jake在我们网站上评价了25首歌曲。



Cooper



Kelsey

Cooper评价了26首歌曲，其中25首和Jake是一样的。他们对每首歌曲的评价差值只有0.25！

Kelsey在我们网站上评价了150首歌曲，其中25首和Jake相同。和Cooper一样，她和Jake之间的评价差值也只有0.25！

所以我们从直觉上看Cooper和Keylsey离Jake的距离应该相似。但是，当我们计算他们之间的曼哈顿距离和欧几里得距离时（代入0值），会发现Cooper要比Keylsey离Jake近得多。

为什么呢？

我们来看下面的数据：

Song:	1	2	3	4	5	6	7	8	9	10
Jake	0	0	0	4.5	5	4.5	0	0	0	0
Cooper	0	0	4	5	5	5	0	0	0	0
Kelsey	5	4	4	5	5	5	5	5	4	4

从4、5、6这三首歌来看，两人离Jake的距离是相同的，但计算出的曼哈顿距离却不这么显示：

$$d_{Cooper,Jake} = (4-0)+(5-4.5)+(5-5)+5-4.5) = 4+0.5+0+0.5 = 5$$

$$d_{Kelsey,Jake} = (5-0)+(4-0)+(4-0)+(5-4.5)+(5-5)+(5-4.5)+(5-0)$$

$$+(5-0)+(4-0)+(4-0)$$

$$= 5+4+4+0.5+0+.5+5+5+4+4 = 32$$

问题就在于数据中的0值对结果的影响很大，所以用0代替空值的方法并不比原来的方程好。还有一种变通的方式是计算“平均值”——将两人共同评价过的歌曲分数除以歌曲数量。

总之，曼哈顿距离和欧几里得距离在数据完整的情况下会运作得非常好，如果数据比较稀疏，则要考虑使用余弦距离。

古怪的现象

假设我们要为Amy推荐乐队，她喜欢Phoenix、Passion Pit、以及Vampire Weekend。和她最相似的用户是Bob，他也喜欢这三支乐队。他的父亲为Walter Ostanek乐队演奏手风琴，所以受此影响，他给了这支乐队5星评价。按照我们现在的推荐逻辑，我们会将这支乐队推荐给Amy，但有可能她并不喜欢。



或者试想一下，Billy Bob Olivera教授喜欢阅读数据挖掘方面的书籍以及科幻小说，他最邻近的用户是我，因为我也喜欢这两种书。然而，我又是一个贵宾犬的爱好者，所以给《贵宾犬的隐秘生活》这本书打了很高的分。这样一来，现有的推荐方法会将这本书介绍给Olivera教授。



问题就在于我们只依靠最相似的 一个 用户来做推荐，如果这个用户有些特殊的偏好，就会直接反映在推荐内容里。解决方法之一是找寻多个相似的用户，这里就要用到K最邻近算法了。

K最邻近算法

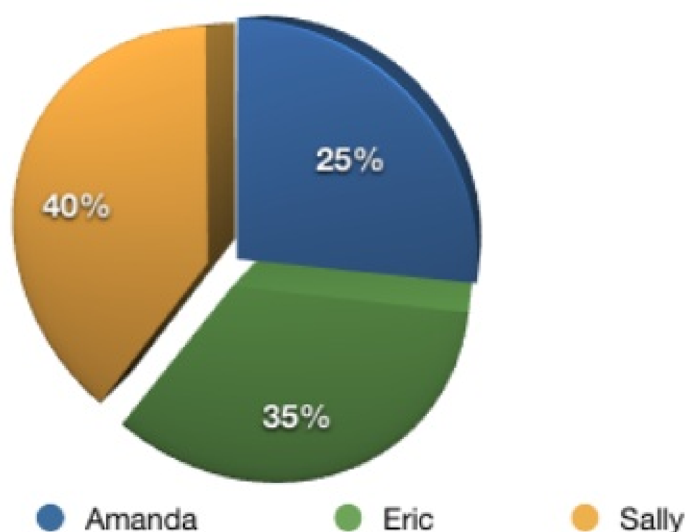
在协同过滤中可以使用K最邻近算法来找出K个最相似的用户，以此作为推荐的基础。不同的应用有不同的K值，需要做一些实验来得出。以下给到读者一个基本的思路。

假设我要为Ann做推荐，并令K=3。使用皮尔逊相关系数得到的结果是：

Person	Pearson
Sally	0.8
Eric	0.7
Amanda	0.5

$$0.8 + 0.7 + 0.5 = 2.0$$

这三个人都会对推荐结果有所贡献，问题在于我们如何确定他们的比重呢？我们直接用相关系数的比重来描述，Sally的比重是0.8/2=40%，Eric是0.7/2=35%，Amanda则是25%：



假设他们三人对Grey Wardens的评分以及加权后的结果如下：

Person	Grey Wardens Rating	Influence
Amanda	4.5	25.00%
Eric	5	35.00%
Sally	3.5	40.00%

最后计算得到的分数为：

$$\text{Projected rating} = (4.5 \times 0.25) + (5 \times 0.35) + (3.5 \times 0.4)$$

$$= 4.275$$

Python推荐模块

我将本章学到的内容都汇集成了一个Python类，虽然代码有些长，我还是贴在了这里：

```
import codecs
from math import sqrt

users = {"Angelica": {"Blues Traveler": 3.5, "Broken Bells": 2.0,
                      "Norah Jones": 4.5, "Phoenix": 5.0,
                      "Slightly Stoopid": 1.5,
                      "The Strokes": 2.5, "Vampire Weekend": 2.0},

         "Bill":{"Blues Traveler": 2.0, "Broken Bells": 3.5,
                 "Deadmau5": 4.0, "Phoenix": 2.0,
                 "Slightly Stoopid": 3.5, "Vampire Weekend": 3.0},

         "Chan": {"Blues Traveler": 5.0, "Broken Bells": 1.0,
                  "Deadmau5": 1.0, "Norah Jones": 3.0, "Phoenix": 5,
                  "Slightly Stoopid": 1.0},

         "Dan": {"Blues Traveler": 3.0, "Broken Bells": 4.0,
                 "Deadmau5": 4.5, "Phoenix": 3.0,
                 "Slightly Stoopid": 4.5, "The Strokes": 4.0,
                 "Vampire Weekend": 2.0},

         "Hailey": {"Broken Bells": 4.0, "Deadmau5": 1.0,
                    "Norah Jones": 4.0, "The Strokes": 4.0,
                    "Vampire Weekend": 1.0},

         "Jordyn":  {"Broken Bells": 4.5, "Deadmau5": 4.0,
                     "Norah Jones": 5.0, "Phoenix": 5.0,
                     "Slightly Stoopid": 4.5, "The Strokes": 4.0,
                     "Vampire Weekend": 4.0},

         "Sam": {"Blues Traveler": 5.0, "Broken Bells": 2.0,
                 "Norah Jones": 3.0, "Phoenix": 5.0,
                 "Slightly Stoopid": 4.0, "The Strokes": 5.0},

         "Veronica": {"Blues Traveler": 3.0, "Norah Jones": 5.0,
                      "Phoenix": 4.0, "Slightly Stoopid": 2.5,
                      "The Strokes": 3.0}
        }


class recommender:

    def __init__(self, data, k=1, metric='pearson', n=5):
        """ 初始化推荐模块
        data   训练数据
        k      K邻近算法中的值
        metric 使用何种距离计算方式
        n      推荐结果的数量
        """
        self.k = k
        self.n = n
        self.username2id = {}
        self.userid2name = {}
        self.productid2name = {}
        # 将距离计算方式保存下来
        self.metric = metric
        if self.metric == 'pearson':
            self.fn = self.pearson
        #
        # 如果data是一个字典类型，则保存下来，否则忽略
        #
        if type(data).__name__ == 'dict':
            self.data = data

    def convertProductID2name(self, id):
        """通过产品ID获取名称"""
        if id in self.productid2name:
            return self.productid2name[id]
        else:
            return id
```

```
    def userRatings(self, id, n):
        """返回该用户评分最高的物品"""
        print ("Ratings for " + self.userid2name[id])
        ratings = self.data[id]
        print(len(ratings))
        ratings = list(ratings.items())
        ratings = [(self.convertProductID2name(k), v)
                   for (k, v) in ratings]
        # 排序并返回结果
        ratings.sort(key=lambda artistTuple: artistTuple[1],
                     reverse = True)
        ratings = ratings[:n]
        for rating in ratings:
            print("%s\t%i" % (rating[0], rating[1]))

    def loadBookDB(self, path=''):
        """加载BX数据集，path是数据文件位置"""
        self.data = {}
        i = 0
        #
        # 将书籍评分数据放入self.data
        #
        f = codecs.open(path + "BX-Book-Ratings.csv", 'r', 'utf8')
        for line in f:
            i += 1
            #separate line into fields
            fields = line.split(';')
            user = fields[0].strip('"')
            book = fields[1].strip('"')
            rating = int(fields[2].strip().strip('"'))
            if user in self.data:
                currentRatings = self.data[user]
            else:
                currentRatings = {}
            currentRatings[book] = rating
            self.data[user] = currentRatings
        f.close()
        #
        # 将书籍信息存入self.productid2name
        # 包括isbn号、书名、作者等
        #
        f = codecs.open(path + "BX-Books.csv", 'r', 'utf8')
        for line in f:
            i += 1
            #separate line into fields
            fields = line.split(';')
            isbn = fields[0].strip('"')
            title = fields[1].strip('"')
            author = fields[2].strip().strip('"')
            title = title + ' by ' + author
            self.productid2name[isbn] = title
        f.close()
        #
        #  将用户信息存入self.userid2name和self.username2id
        #
        f = codecs.open(path + "BX-Users.csv", 'r', 'utf8')
        for line in f:
            i += 1
            #print(line)
            #separate line into fields
            fields = line.split(';')
            userid = fields[0].strip('"')
            location = fields[1].strip('"')
            if len(fields) > 3:
                age = fields[2].strip().strip('"')
            else:
                age = 'NULL'
            if age != 'NULL':
                value = location + '  (age: ' + age + ')'
            else:
                value = location
            self.userid2name[userid] = value
```

```
                self.username2id[location] = userid
            f.close()
            print(i)

    def pearson(self, rating1, rating2):
        sum_xy = 0
        sum_x = 0
        sum_y = 0
        sum_x2 = 0
        sum_y2 = 0
        n = 0
        for key in rating1:
            if key in rating2:
                n += 1
                x = rating1[key]
                y = rating2[key]
                sum_xy += x * y
                sum_x += x
                sum_y += y
                sum_x2 += pow(x, 2)
                sum_y2 += pow(y, 2)
        if n == 0:
            return 0
        # 计算分母
        denominator = (sqrt(sum_x2 - pow(sum_x, 2) / n)
                       * sqrt(sum_y2 - pow(sum_y, 2) / n))
        if denominator == 0:
            return 0
        else:
            return (sum_xy - (sum_x * sum_y) / n) / denominator

    def computeNearestNeighbor(self, username):
        """获取邻近用户"""
        distances = []
        for instance in self.data:
            if instance != username:
                distance = self.fn(self.data[username],
                                   self.data[instance])
                distances.append((instance, distance))
        # 按距离排序，距离近的排在前面
        distances.sort(key=lambda artistTuple: artistTuple[1],
                       reverse=True)
        return distances

    def recommend(self, user):
       """返回推荐列表"""
       recommendations = {}
       # 首先，获取邻近用户
       nearest = self.computeNearestNeighbor(user)
       #
       # 获取用户评价过的商品
       #
       userRatings = self.data[user]
       #
       # 计算总距离
       totalDistance = 0.0
       for i in range(self.k):
          totalDistance += nearest[i][1]
       # 汇总K邻近用户的评分
       for i in range(self.k):
          # 计算饼图的每个分片
          weight = nearest[i][1] / totalDistance
          # 获取用户名称
          name = nearest[i][0]
          # 获取用户评分
          neighborRatings = self.data[name]
          # 获得没有评价过的商品
          for artist in neighborRatings:
             if not artist in userRatings:
                if artist not in recommendations:
                   recommendations[artist] = (neighborRatings[artist]
                                              * weight)
```

```
                else:
                    recommendations[artist] = (recommendations[artist]
                                               + neighborRatings[artist]
                                               * weight)
        # 开始推荐
        recommendations = list(recommendations.items())
        recommendations = [(self.convertProductID2name(k), v)
                           for (k, v) in recommendations]
        # 排序并返回
        recommendations.sort(key=lambda artistTuple: artistTuple[1],
                             reverse = True)
        # 返回前n个结果
        return recommendations[:self.n]
```

运行示例

首先构建一个推荐类，然后获取推荐结果：

```
>>> r = recommender(users)
>>> r.recommend('Jordyn')
[('Blues Traveler', 5.0)]
>>> r.recommend('Hailey')
[('Phoenix', 5.0), ('Slightly Stoopid', 4.5)]
```

新的数据集

现在让我们使用一个更为真实的数据集。Cai-Nicolas Zeigler从图书漂流站收集了超过100万条评价数据——278,858位用户为271,379本书打了分。这份数据（匿名）可以从这个地址获得，有SQL和CSV两种格式。由于特殊符号的关系，这些数据无法直接加载到Python里。我做了一些清洗，可以从这里下载。

CSV文件包含了三张表：

- 用户表，包括用户ID、位置、年龄等信息。其中用户的姓名已经隐去；
- 书籍表，包括ISBN号、标题、作者、出版日期、出版社等；
- 评分表，包括用户ID、书籍ISBN号、以及评分（0-10分）。

上文Python代码中的loadBookDB方法可以加载这些数据，用法如下：

```
>>> r.loadBookDB('/Users/raz/Downloads/BX-Dump/')
1700018
>>> r.recommend('171118')
```

注意 由于数据集比较大，大约需要几十秒的时间加载和查询。

项目实践

只有运行调试过书中的代码后才能真正掌握这些方法，以下是一些实践建议：

1. 实现一个计算曼哈顿距离和欧几里得距离的方法；
2. 本书的网站上有一个包含25部电影评价的数据集，实现一个推荐算法。

第三章：隐式评价和基于物品的过滤算法

原文：http://guidetodatamining.com/chapter-3/

本章会从用户的评价类型开始讨论，包括显式评价（赞一下、踩一脚、五星评价等等）和隐式评价（比如在亚马逊上购买了MP3，我们可以认为他喜欢这个产品）。

内容：

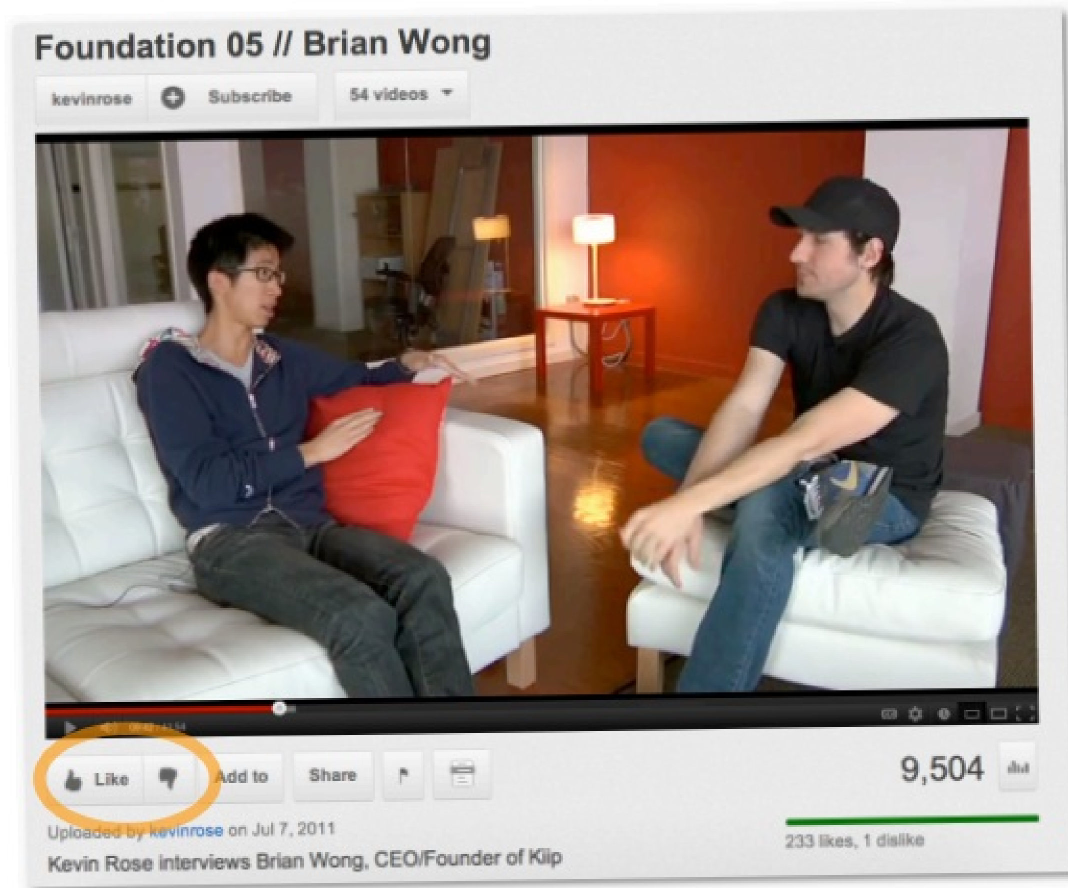
- 显式评价
- 隐式评价
- 哪种评价方式更准确？
- 基于用户的协同过滤
- 基于物品的协同过滤
- 修正的余弦相似度
- Slope One算法
- Slope One的Python实现
- MovieLens数据

第二章中我们学习了协同过滤和推荐系统的基本知识，其中讲述的算法是比较通用的，可以适用于多种数据集。用户使用5到10分的标尺来对不同的物品进行打分，通过计算得到相似的用户。但是，也有迹象表明用户通常不会有效地使用这种度量方式，而更倾向于给出极好或极差的评价，这种做法会使推荐结果变得不可用。这一章我们将继续探讨这个问题，尝试使用高效的方法给出更精确的推荐。

显式评价

用户的评价类型可以分为显式评价和隐式评价。显式评价指的是用户明确地给出对物品的评价，最常见的例子是Pandora和YouTube上的“喜欢”和“不喜欢”按钮：





以及亚马逊的星级系统：



隐式评价

所谓隐式评价，就是我们不让用户明确给出对物品的评价，而是通过观察他们的行为来获得偏好信息。示例之一是记录用户在纽约时报网上的点击记录。



经过几周的观察之后，我们就可以为用户刻画出一个合理的模型了——她不喜欢体育新闻，但关注科技新闻；如果用户连续看了两篇文章：《快速减肥方法》和《不反弹的减肥方式》，那她很可能正在减肥；如果她点击了iPhone的广告，就表明她或许对这款产品感兴趣。

试想一下，如果我们记录了用户在亚马逊上的操作记录，可以得出一些什么结论。你的首页上可能有这样的内容：

More Items to Consider

You viewed | Customers who viewed this also viewed

Jupiters Travels: Four Years Around...
› Ted Simon
Paperback
(65)
$24.95 $16.47

Long Way Round
Ewan McGregor, Charley Boorman, ...
DVD
(325)
$24.95 $16.93

One More Day Everywhere: Crossing 50...
› Glen Heggstad
Paperback
(47)
$18.95 $13.87

Dreaming of Jupiter
› Ted Simon
Paperback
(6)
$16.22

在这个示例中，亚马逊记录了用户的点击操作，因此它会知道浏览了Jupter Travel这本书的用户还浏览了Long Way Round这部DVD，其详细记录了演员伊万环球骑行的旅程。因此，亚马逊就用这些信息来做出“看过还看过”的推荐。

另一种隐式评价是用户的实际购买记录，亚马逊也会用这些记录来进行“买过还买过”、以及“看过此商品的用户还买过”的推荐。

Frequently Bought Together

Price For All Three: $41.87

Add all three to Cart Add all three to Wish List

Show availability and shipping details

This item: One More Day Everywhere: Crossing 50 Borders on the Road to Global Understanding by Glen Heggstad Paperback $13.87

Two Wheels Through Terror: Diary of a South American Motorcycle Odyssey by Glen Heggstad Paperback $11.53

Jupiters Travels: Four Years Around the World on a Triumph by Ted Simon Paperback $16.47

Customers Who Viewed This Item Also Bought

Jupiters Travels: Four Years Around the World on ... › Ted Simon (65) Paperback $16.47

Two Wheels Through Terror: Diary of a South American ... › Glen Heggstad (50) Paperback $11.53

可能你会觉得“买过还买过”应该会给出一些不合理的推荐结果，但事实上它运作得很好。

再来看看iTunes上如何记录用户的行为：

Name	Time	Artist	Plays
Anchor	3:24	Zee Avi	52
My Companjera	3:22	Gogol Bordello	27
Wake Up Everybody	4:25	John Legend & the...	17
Milestone Moon	3:40	Zee Avi	17
...			

首先，我将一首歌添加到了iTunes，这至少表明我对这首歌是感兴趣的。然后是播放次数，上表中我听了Anchor这首歌52次，说明我很喜欢；而那些只听了一次的歌曲则是我不喜欢的。

头脑风暴

> 你觉得让用户对物品进行显式评价会更精确吗？还是说通过观察用户对物品的行为（是否购买或播放次数）才更为准确？



> 显式评价：我叫吉姆，是一个素食主义者。我爱喝葡萄酒，喜欢在森林中漫步，在篝火旁阅读Chekov的书，喜欢观看法国电影，周六会去艺术博物馆逛逛，我还喜欢舒曼的钢琴曲。



> 隐式评价：我们在吉姆的口袋里发现了12打美国蓝带啤酒的收银条，以及冰激淋、披萨和甜甜圈的收银条。还有一些租借DVD的回执，有复仇者联盟、生化危机、拳霸等。

显式评价的问题

问题1：人们很懒，不愿评价物品

首先，用户很可能不会对物品做出评价。相信各位读者已经在亚马逊上购买了很多商品，就拿我来说，仅过去一个月我就在那里购买了直升机模型、1TB硬盘、USB-SATA转接头、维他命药片、两本Kindle电子书、四本纸质书。一共十件商品，我评价了几件？零件！相信很多人和我是一样的——我们不评价商品，我们只管买。

我喜欢旅行和登山，所以购买了很多登山杖。亚马逊上一些价格实惠的登山杖很耐用。去年我到奥斯汀市参加音乐会，途中碰坏了膝盖，于是到REI专营店买了一根价格昂贵的登山杖。不过这根杖居然在我逛公园时用断了！这根昂贵的登山杖还没有买的10美元的来得结实。放假时，我打算给这件商品写一篇评价，告诫其他购买者。结果呢？我没有写，因为我太懒了。

问题2：人们会撒谎，或存有偏见

我们假设有人不像前面说得那么懒，确实去给物品做出评价了，但他有可能会撒谎。这种情况在前文中已经有提到了。用户可能会直接撒谎，给出不正确的评价；或是不置可否，抱有偏见。Ben和他的朋友们去看了一场泰国出的电影，Ben认为这部电影很糟糕，而其他人却觉得很好看，在餐厅里欢快地谈论着。于是，Ben在评价电影时很有可能会抬高它的分数，这样才能表现得合群。

问题3：人们不会更新他们的评论

假设我去亚马逊评价了商品——那个1TB的硬盘速度很快也很静音；直升机模型操作起来也很简便，不容易摔坏。所以这两件商品我都给出了5星的评价。但一个月后，那块硬盘坏了，我丢失了所有的电影和音乐；那台直升机模型也突然不再工作了，让我非常扫兴。但是，我不太会返回亚马逊网站对这两件商品的评价做出改动，这样人们依旧认为我是非常喜欢这两件商品的。



再举一个示例，玛丽很乐意在亚马逊上对商品做评价。她十年前给一些儿童类书籍打了很高的分数，近些年又对一些摇滚乐队的专辑给出了评价。从近年的评价看，她和另一位用户珍妮很相似。但是，如果我们把那些儿童书籍推荐给珍妮就显得不合适了。这个例子和上面的有些不同，但的确是个问题。

头脑风暴

> 你觉得隐式评价会有什么问题？提示：可以回忆一下你在亚马逊的购买记录。

上文中我给出了一个近期在亚马逊上的购物列表，其中有两样是我买来送给其他人的。为什么这会是一个问题？我再举一些其他的例子。我给我的孩子买了一个壶铃和一本关于健身的书籍；我给我的太太买了一个边境牧羊犬的毛绒玩具，因为我家那只14岁大的狗去世了。通过隐式评价来进行建模，会让你觉得我喜欢壶铃和毛绒玩具。亚马逊的购买记录无法区分这件商品是我买来自己用的还是送人的。贝克也曾给出了相似的例子：



Baker 2008.60-61.

> 对于计算机来说，能够将白色连衣裙和婴儿潮出生的女性关联起来是任务的第一步，然后再对这些用户建立模型。假设我的太太在商店里购买了几件商品：内衣、裤子、连衣裙、皮带等，这些商品都很符合婴儿潮的特点。离开时她想起要为自己16岁大的外甥女买一件生日礼物。由于我们上次看到她时她穿着一件黑色的T恤，上面写满了文字，并自称是一名哥特摇滚妞。于是，我的太太就去买了一根项圈准备送给她。

可以想象，如果我们要为这位用户构建模型，那这根项圈的存在就很有问题了。

再比如一对情侣使用的是同一个Netflix账号。男方喜欢各种爆破场面，女方则喜欢知性类型的电影。如果我们从浏览历史进行挖掘，则会发现一个人会喜欢两种截然不同的影片类型。

前面说到我买了一些书给别人，所以单从购买历史看，同一本书我会购买很多次。这样有两种可能：一是我的书不小心丢了，二是我得了老年痴呆，不记得自己曾读过这些书。而事实是我非常喜欢这些书，因此多买了几本作为礼物来送给别人。所以说，用户的购买记录还是非常值得深究的。

头脑风暴

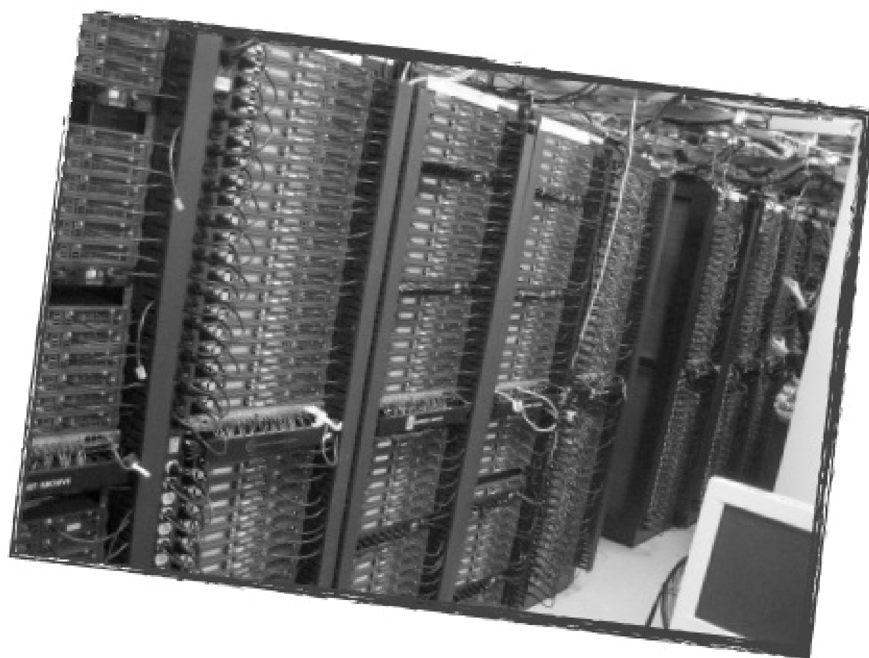
> 我们可以收集到哪些隐式评价呢？ 网页方面：页面点击、停留时间、重复访问次数、引用率、Hulu上观看视频的次数； 音乐播放器：播放的曲目、跳过的曲目、播放次数； 这些只是一小部分！

值得注意的是，我们在第二章中学习的算法对于显式评价和隐式评价都是适用的。

什么会阻碍你成功？

设想你有一个成熟的在线音乐网站，在构建推荐系统时会遇到什么问题呢？

假设你有一百万个用户，每次推荐需要计算一百万个距离数据。如果我们想在一秒钟里进行多次推荐，那计算量将是巨大的。除非增加服务器的数量，否则系统会变得越来越慢。说得专业一点，通过邻域进行计算的推荐系统，延迟会变得越来越严重。还好，这是有解决办法的。



基于用户的协同过滤

目前为止我们描述的都是基于用户的协同过滤算法。我们将一个用户和其他所有用户进行对比，找到相似的人。这种算法有两个弊端：

1. 扩展性 上文已经提到，随着用户数量的增加，其计算量也会增加。这种算法在只有几千个用户的情况下能够工作得很好，但达到一百万个用户时就会出现瓶颈。
2. 稀疏性 大多数推荐系统中，物品的数量要远大于用户的数量，因此用户仅仅对一小部分物品进行了评价，这就造成了数据的稀疏性。比如亚马逊有上百万本书，但用户只评论了很少一部分，于是就很难找到两个相似的用户了。

鉴于以上两个局限性，我们不妨考察一下基于物品的协同过滤算法。

基于物品的协同过滤

假设我们有一种算法可以计算出两件物品之间的相似度，比如Phoenix专辑和Manners很相似。如果一个用户给Phoenix打了很高的分数，我们就可以向他推荐Manners了。需要注意这两种算法的区别：基于用户的协同过滤是通过计算用户之间的距离找出最相似的用户，并将他评价过的物品推荐给目标用户；而基于物品的协同过滤则是找出最相似的物品，再结合用户的评价来给出推荐结果。

能否举个例子？

我们的音乐站点有m个用户和n个乐队，用户会对乐队做出评价，如下表所示：

	Users	...	Phoenix	...	Passion Pit	...	n
1	Tamera Young		5				
2	Jasmine Abbey				4		
3	Arturo Alvarez		1		2		
...	...						
u	Cecilia De La Cueva		5		5		
...	...						
m-1	Jessica Nguyen		4		5		
m	Jordyn Zamora		4				

我们要计算Phoenix和Passion Pit之间的相似度，可以使用蓝色方框中的数据，也就是同时对这两件商品都有过评价的用户。在基于用户的算法中，我们计算的是行与行之间的相似度，而在基于物品的算法中，我们计算的是列与列之间的。





> 基于用户的协同过滤又称为内存型协同过滤，因为我们需要将所有的评价数据都保存在内存中来进行推荐。
>
> 基于物品的协同过滤也称为基于模型的协同过滤，因为我们不需要保存所有的评价数据，而是通过构建一个物品相似度模型来做推荐。

修正的余弦相似度

我们使用余弦相似度来计算两个物品的距离。我们在第二章中提过“分数膨胀”现象，因此我们会从用户的评价中减去他所有评价的均值，这就是修正的余弦相似度。



> 左：我喜欢Phoenix乐队，因此给他们打了5分。我不喜欢Passion，所以给了3分。
>
> 右：Phoenix很棒，我给4分。Passion Pit太糟糕了，必须给0分！

$$s(i,j)=\frac{\sum_{u\in U}(R_{u,i}-\overline{R}_u)(R_{u,j}-\overline{R}_u)}{\sqrt{\sum_{u\in U}(R_{u,i}-\overline{R}_u)^2}\sqrt{\sum_{u\in U}(R_{u,j}-\overline{R}_u)^2}}$$

*U*表示同时评价过物品*i*和*j*的用户集合

这个公式来自于一篇影响深远的论文《基于物品的协同过滤算法》，由Badrul Sarwar等人合著。

$$\left(R_{u,i}-\overline{R}_u\right)$$

上式表示将用户u对物品i的评价值减去用户u对所有物品的评价均值，从而得到修正后的评分。s(i,j)表示物品i和j的相似度，分子表示将同时评价过物品i和j的用户的修正评分相乘并求和，分母则是对所有的物品的修正评分做一些汇总处理。

为了更好地演示修正的余弦相似度，我们举一个例子。下表是五个学生对五位歌手的评价：

Users	average rating	Kacey Musgraves	Imagine Dragons	Daft Punk	Lorde	Fall Out Boy
David			3	5	4	1
Matt			3	4	4	1
Ben		4	3		3	1
Chris		4	4	4	3	1
Torri		5	4	5		3

首先，我们计算出每个用户的平均评分，这很简单：

Users	average rating	Kacey Musgraves	Imagine Dragons	Daft Punk	Lorde	Fall Out Boy
David	3.25		3	5	4	1
Matt	3.0		3	4	4	1
Ben	2.75	4	3		3	1
Chris	3.2	4	4	4	3	1
Tori	4.25	5	4	5		3

下面，我们计算歌手之间的相似度，从Kacey Musgraves和Imagine Dragons开始。上图中我已经标出了同时评价过这两个歌手的用户，代入到公式中：

$$s(Musgraves, Dragons) = \frac{\sum_{u \in U}(R_{u,Musgraves} - \overline{R}_u)(R_{u,Dragons} - \overline{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,Musgraves} - \overline{R}_u)^2}\sqrt{\sum_{u \in U}(R_{u,Dragons} - \overline{R}_u)^2}}$$

Ben's ratings | Chris's ratings | Tori's ratings

$$= \frac{(4-2.75)(3-2.75)+(4-3.2)(4-3.2)+(5-4.25)(4-4.25)}{\sqrt{(4-2.75)^2+(4-3.2)^2+(5-4.25)^2}\sqrt{(3-2.75)^2+(4-3.2)^2+(4-4.25)^2}}$$

$$= \frac{0.7650}{\sqrt{2.765}\sqrt{0.765}} = \frac{0.7650}{(1.6628)(0.8746)} = \frac{0.7650}{1.4543} = 0.5260$$

所以这两个歌手之间的修正余弦相似度为0.5260，我计算了其他一些歌手之间的相似度，其余的请读者们完成：

	Fall Out Boy	Lorde	Daft Punk	Imagine Dragons
Kacey Musgraves	-0.9549		1.0000	0.5260
Imagine Dragons	-0.3378		0.0075	
Daft Punk	-0.9570			
Lorde	-0.6934			
Fall Out Boy				

计算修正余弦相似度的Python代码

```
# -*- coding: utf-8 -*-

from math import sqrt

users3 = {"David": {"Imagine Dragons": 3, "Daft Punk": 5,
                    "Lorde": 4, "Fall Out Boy": 1},
          "Matt": {"Imagine Dragons": 3, "Daft Punk": 4,
                   "Lorde": 4, "Fall Out Boy": 1},
          "Ben": {"Kacey Musgraves": 4, "Imagine Dragons": 3,
                  "Lorde": 3, "Fall Out Boy": 1},
          "Chris": {"Kacey Musgraves": 4, "Imagine Dragons": 4,
                    "Daft Punk": 4, "Lorde": 3, "Fall Out Boy": 1},
          "Tori": {"Kacey Musgraves": 5, "Imagine Dragons": 4,
                   "Daft Punk": 5, "Fall Out Boy": 3}}


def computeSimilarity(band1, band2, userRatings):
    averages = {}
    for (key, ratings) in userRatings.items():
        averages[key] = (float(sum(ratings.values())) / len(ratings.values()))

    num = 0 # 分子
    dem1 = 0 # 分母的第一部分
    dem2 = 0
    for (user, ratings) in userRatings.items():
        if band1 in ratings and band2 in ratings:
            avg = averages[user]
            num += (ratings[band1] - avg) * (ratings[band2] - avg)
            dem1 += (ratings[band1] - avg) ** 2
            dem2 += (ratings[band2] - avg) ** 2
    return num / (sqrt(dem1) * sqrt(dem2))

print computeSimilarity('Kacey Musgraves', 'Lorde', users3)
print computeSimilarity('Imagine Dragons', 'Lorde', users3)
print computeSimilarity('Daft Punk', 'Lorde', users3)
```

	Fall Out Boy	Lorde	Daft Punk	Imagine Dragons
Kacey Musgraves	-0.9549	0.3210	1.0000	0.5260
Imagine Dragons	-0.3378	-0.2525	0.0075	
Daft Punk	-0.9570	0.7841		
Lorde	-0.6934			



> 这个矩阵看起来不错，那下面该如何使用它来做预测呢？比如我想知道David有多喜欢Kacey Musgraves？

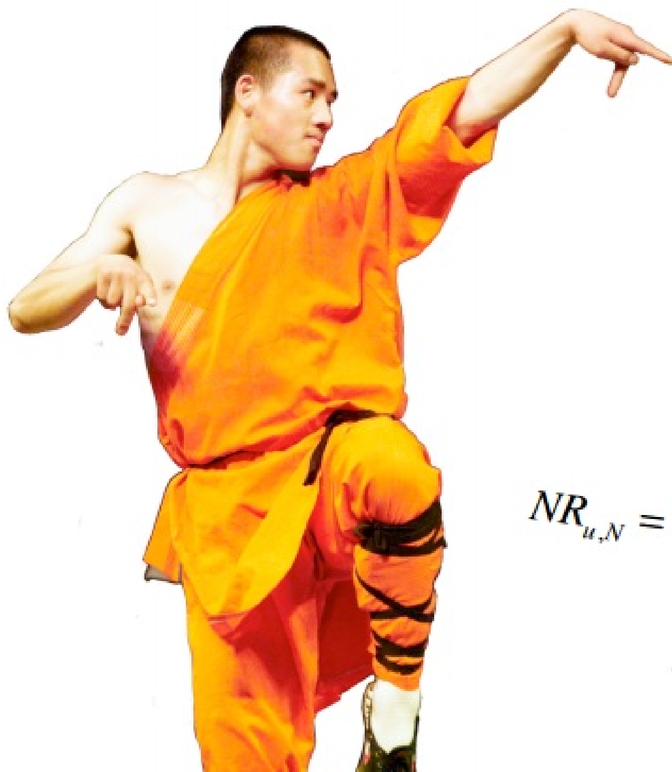
$$p(u,i)=\frac{\sum_{N\in similarTo(i)}(S_{i,N}\times R_{u,N})}{\sum_{N\in similarTo(i)}(|S_{i,N}|)}$$

p(u,i)表示我们会来预测用户u对物品i的评分，所以p(David, Kacey Musgraves)就表示我们将预测David会给Kacey打多少分。

N是一个物品的集合，有如下特性：用户u对集合中的物品打过分；物品i和集合中的物品有相似度数据（即上文中的矩阵）。

$S_{i,N}$表示物品i和N的相似度，$R_{u,N}$表示用户u对物品N的评分。

为了让公式的计算效果更佳，对物品的评价分值最好介于-1和1之间。由于我们的评分系统是1至5星，所以需要使用一些运算将其转换到-1至1之间。



我们的音乐评分系统是5分制，Max_R表示评分系统中的最高分（这里是5），Min_R为最低分（这里是1），$R_{u,N}$是用户u对物品N的评分，$NR_{u,N}$则表示修正后的评分（即范围在-1和1之间）。

若已知$NR_{u,N}$，求解$R_{u,N}$的公式为：

$$R_{u,N} = \frac{1}{2}((NR_{u,N} + 1) \times (Max_R - Min_R)) + Min_R$$

比如一位用户给Fall Out Boy打了2分，那修正后的评分为：

$$NR_{u,N} = \frac{2(R_{u,N} - Min_R) - (Max_R - Min_R)}{(Max_R - Min_R)} = \frac{2(2-1) - (5-1)}{(5-1)} = \frac{-2}{4} = -0.5$$

反过来则是：

$$R_{u,N} = \frac{1}{2}((NR_{u,N} + 1) \times (Max_R - Min_R)) + Min_R$$

$$= \frac{1}{2}((-0.5 + 1) \times 4) + 1 = \frac{1}{2}(2) + 1 = 1 + 1 = 2$$

有了这个基础后，下面就让我们看看如何求解上文中的p(David, Kacey Musgraves)。

首先我们要修正David对各个物品的评分：

Artist	R	NR
Imagine Dragons	3	0
Daft Punk	5	1
Lorde	4	0.5
Fall Out Boy	1	-1

然后结合物品相似度矩阵，代入公式：

$$p(u,i) = \frac{\sum_{N \in similarTo(i)} (S_{i,N} \times NR_{u,N})}{\sum_{N \in similarTo(i)} (|S_{i,N}|)} =$$

Imagine Dragons　Daft Punk　Lorde　Fall Out Boy

$$\frac{(.5260 \times 0) + (1.00 \times 1) + (.321 \times 0.5) + (-.955 \times -1)}{0.5260 + 1.000 + 0.321 + 0.955}$$

$$= \frac{0 + 1 + 0.1605 + 0.955}{2.802} = \frac{2.1105}{2.802} = 0.753$$

所以，我们预测出David对Kacey Musgraves的评分是0.753，将其转换到5星评价体系中：

$$R_{u,N} = \frac{1}{2}((NR_{u,N} + 1) \times (Max_R - Min_R)) + Min_R$$

$$= \frac{1}{2}((0.753 + 1) \times 4) + 1 = \frac{1}{2}(7.012) + 1 = 3.506 + 1 = 4.506$$

最终的预测结果是4.506分。

回顾

- 修正的余弦相似度是一种基于模型的协同过滤算法。我们前面提过，这种算法的优势之

一是扩展性好，对于大数据量而言，运算速度快、占用内存少。
- 用户的评价标准是不同的，比如喜欢一个歌手时有些人会打4分，有些打5分；不喜欢时有人会打3分，有些则会只给1分。修正的余弦相似度计算时会将用户对物品的评分减去用户所有评分的均值，从而解决这个问题。

Slope One算法

还有一种比较流行的基于物品的协同过滤算法，名为Slope One，它最大的优势是简单，因此易于实现。Slope One算法是在一篇名为《Slope One：基于在线评分系统的协同过滤算法》的论文中提出的，由Lemire和Machlachlan合著。这篇论文非常值得一读。

我们用一个简单的例子来了解这个算法。假设Amy给PSY打了3分，Whitney Houston打了4分；Ben给PSY打了4分。我们要预测Ben会给Whitney Houston打几分。用表格来描述这个问题即：

	PSY	Whitney Houston
Amy	3	4
Ben	4	?

我们可以用以下逻辑来预测Ben对Whitney Houston的评分：由于Amy给Whitney Houston打的分数要比PSY的高一分，所以我们预测Ben也会给高一分，即给到5分。

其实还有其他形式的Slope One算法，比如加权的Slope One。我们说过Slope One的优势之一是简单，下面说的加权的Slope One看起来会有一些复杂，但是只要耐心地看下去，事情就会变得很清晰了。

你可以将Slope One分为两个步骤：首先需要计算出两两物品之间的差值（可以在夜间批量计算）。在上文的例子中，这个步骤就是得出Whitney Houston要比PSY高一分。第二步则是进行预测，比如一个新用户Ben来到了我们网站，他从未听过Whitney Houston的歌曲，我们想要预测他是否喜欢这位歌手。通过利用他评价过的歌手以及我们计算好的歌手之间的评分差值，就可以进行预测了。

第一步：计算差值

我们先为上述例子增加一些数据：

	Taylor Swift	PSY	Whitney Houston
Amy	4	3	4
Ben	5	2	?
Clara	?	3.5	4
Daisy	5	?	3

计算物品之间差异的公式是：

$$dev_{i,j} = \sum_{u \in S_{i,j}(X)} \frac{u_i - u_j}{card(S_{i,j}(X))}$$

其中，card(S)表示S中有多少个元素；X表示所有评分值的集合；card($S_{j,i}$(X))则表示同时评价过物品j和i的用户数。我们来考察PSY和Taylor Swift之间的差值，card($S_{j,i}$(X))的值是2——因为有两个用户（Amy和Ben）同时对PSY和Taylor Swift打过分。分子u_j-u_i表示用户对j的评分减去对i的评分，代入公式得：

$$dev_{swift,psy} = \frac{(4-3)}{2} + \frac{(5-2)}{2} = \frac{1}{2} + \frac{3}{2} = 2$$

所以PSY和Taylor Swift的差异是2，即用户们给Taylor Swift的评分比PSY要平均高出两分。那Taylor Swift和PSY的差异呢？

$$dev_{psy,swift} = \frac{(3-4)}{2} + \frac{(2-5)}{2} = -\frac{1}{2} + -\frac{3}{2} = -2$$

作业：计算其他物品之间的差值

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2	1
PSY	-2	0	-0.75
Whitney Houston	-1	0.75	0

头脑风暴

试想我们的音乐站点有100万个用户对20万个歌手做评价。如果有一个新进的用户对10个歌手做了评价，我们是否需要重新计算20万×20万的差异数据，或是有其他更简单的方法？



答案是你不需要计算整个数据集，这正是Slope One的美妙之处。对于两个物品，我们只需记录同时评价过这对物品的用户数就可以了。比如说Taylor Swift和PSY的差值是2，是根据9位用户的评价计算的。当有一个新用户对Taylor Swift打了5分，PSY打了1分时，更新后的差值为：

$$((9 * 2) + 4) / 10 = 2.2$$



第二步：使用加权的Slope One算法进行预测

好，现在我们有了物品之间的差异值，下面就用它来进行预测。这里我们将使用加权的Slope One算法来进行预测，用P^{WS1}来表示，公式为：

$$P^{wS1}(u)_j = \frac{\sum_{i \in S(u)-\{j\}} (dev_{j,i} + u_i) c_{j,i}}{\sum_{i \in S(u)-\{j\}} c_{j,i}}$$

其中：

$$c_{j,i} = card(S_{j,i}(\chi))$$

$P^{WS1}(u)_j$表示我们将预测用户u对物品i的评分。比如$P^{WS1}(Ben)_{Whitney\ Houston}$表示Ben对Whitney Houston的预测评分。下面就让我们来求解这个问题。

首先来看分子：

$$\sum_{i \in S(u)-\{j\}}$$

表示遍历Ben评价过的所有歌手，除了Whitney Houston以外（也就是-{j}的意思）。

整个分子的意思是：对于Ben评价过的所有歌手（Whitney Houston除外），找出Whitney Houston和这些歌手之间的差值，并将差值加上Ben对这个歌手的评分。同时，我们要将这个结果乘以同时评价过两位歌手的用户数。

让我们分解开来看，先将Ben的评分情况和两两歌手之间的差异值展示如下：

	Taylor Swift	PSY	Whitney Houston
Ben	5	2	?

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2	1
PSY	-2	0	-0.75
Whitney Houston	-1	0.75	0

1. Ben对Taylor Swift打了5分，也就是u_i
2. Whitney Houston和Taylor Swift的差异是-1，即$dev_{j,i}$
3. $dev_{j,i} + u_i = 4$
4. 共有两个用户（Amy和Daisy）同时对Taylor Swift和Whitney Houston做了评价，即$c_{j,i} = 2$
5. 那么$(dev_{j,i} + u_i)\ c_{j,i} = 4 \times 2 = 8$

6. Ben对PSY打了2分
7. Whitney Houston和PSY的差异是0.75
8. $dev_{j,i} + u_i = 2.75$
9. 有两个用户同时评价了这两位歌手，因此$(dev_{j,i} + u_i) c_{j,i} = 2.75 \times 2 = 5.5$
10. 分子：8 + 5.5 = 13.5
11. 分母：2 + 2 = 4
12. 预测评分：13.5 ÷ 4 = 3.375



使用Python实现Slope One算法

我们将沿用第二章中编写的Python类，重复的代码我不在这里赘述。输入的数据是这样的：

```
users2 = {"Amy": {"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4},
          "Ben": {"Taylor Swift": 5, "PSY": 2},
          "Clara": {"PSY": 3.5, "Whitney Houston": 4},
          "Daisy": {"Taylor Swift": 5, "Whitney Houston": 3}}
```

我们先来计算两两物品之间的差异，公式是：

$$dev_{i,j} = \sum_{u \in S_{i,j}(X)} \frac{u_i - u_j}{card(S_{i,j}(X))}$$

计算后的输出结果应该如下表所示：

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2 (2)	1 (2)
PSY	-2 (2)	0	-0.75 (2)
Whitney Houston	-1 (2)	0.75 (2)	0

括号中的数值表示同时给这两个歌手评过分的用户数。

第一步

```
def computeDeviations(self):
    # 获取每位用户的评分数据
    for ratings in self.data.values():
```

self.data是一个Python字典，它的values()方法可以获取所有键的值。比如上述代码在第一次迭代时，ratings变量的值为{"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4}。

第二步

```
def computeDeviations(self):
    # 获取每位用户的评分数据
    for ratings in self.data.values():
        # 对于该用户的每个评分项（歌手、分数）
        for (item, rating) in ratings.items():
            self.frequencies.setdefault(item, {})
            self.deviations.setdefault(item, {})
```

在这个类的初始化方法中，我们需要对self.frequencies和self.deviations进行赋值：

```
def __init__(self, data, k=1, metric='pearson', n=5):
    ...
    # 以下变量将用于Slope One算法
    self.frequencies = {}
    self.deviations = {}
```

Python字典的setdefault()方法接受两个参数，它的作用是：如果字典中不包含指定的键，则将其设为默认值；若存在，则返回其对应的值。

第三步

```
def computeDeviations(self):
    # 获取每位用户的评分数据
    for ratings in self.data.values():
        # 对于该用户的每个评分项（歌手、分数）
        for (item, rating) in ratings.items():
            self.frequencies.setdefault(item, {})
            self.deviations.setdefault(item, {})
            # 再次遍历该用户的每个评分项
            for (item2, rating2) in ratings.items():
                if item != item2:
                    # 将评分的差异保存到变量中
                    self.frequencies[item].setdefault(item2, 0)
                    self.deviations[item].setdefault(item2, 0.0)
                    self.frequencies[item][item2] += 1
                    self.deviations[item][item2] += rating - rating2
```

还是用{"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4}举例，在第一次遍历中，外层循环item = "Taylor Swift"，rating = 4；内层循环item2 = "PSY"，rating2 = 3，因此最后一行代码是对self.deviations["Taylor Swift"]["PSY"]做+1的操作。

第四步

最后，我们便可计算出差异值：

```
def computeDeviations(self):
    # 获取每位用户的评分数据
    for ratings in self.data.values():
        # 对于该用户的每个评分项（歌手、分数）
        for (item, rating) in ratings.items():
            self.frequencies.setdefault(item, {})
            self.deviations.setdefault(item, {})
            # 再次遍历该用户的每个评分项
            for (item2, rating2) in ratings.items():
                if item != item2:
                    # 将评分的差异保存到变量中
                    self.frequencies[item].setdefault(item2, 0)
                    self.deviations[item].setdefault(item2, 0.0)
                    self.frequencies[item][item2] += 1
                    self.deviations[item][item2] += rating - rating2

    for (item, ratings) in self.deviations.items():
        for item2 in ratings:
            ratings[item2] /= self.frequencies[item][item2]
```

完成了！仅仅用了18代码我们就实现了这个公式：

$$dev_{i,j} = \sum_{u \in S_{i,j}(X)} \frac{u_i - u_j}{card(S_{i,j}(X))}$$

让我们测试一下：

```
>>> r = Recommender()
>>> r.computeDeviations()
>>> r.deviations
{'PSY': {'Taylor Swift': -2.0, 'Whitney Houston': -0.75}, 'Taylor Swift': {'PSY': 2.0, 'W
```

结果和我们之前手工计算的一致：

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2	1
PSY	-2	0	-0.75
Whitney Houston	-1	0.75	0

感谢Bryan O'Sullivan，这里用Python实现的Slope One算法正是基于他的成果。



加权的Slope One算法：推荐逻辑的实现

$$P^{wS1}(u)_j = \frac{\sum_{i \in S(u)-\{j\}} (dev_{j,i} + u_i) c_{j,i}}{\sum_{i \in S(u)-\{j\}} c_{j,i}}$$

```
def slopeOneRecommendations(self, userRatings):
    recommendations = {}
    frequencies = {}
    # 遍历目标用户的评分项（歌手、分数）
    for (userItem, userRating) in userRatings.items():
        # 对目标用户未评价的歌手进行计算
        for (diffItem, diffRatings) in self.deviations.items():
            if diffItem not in userRatings and userItem in self.deviations[diffItem]:
                freq = self.frequencies[diffItem][userItem]
                recommendations.setdefault(diffItem, 0.0)
                frequencies.setdefault(diffItem, 0)
                # 分子
                recommendations[diffItem] += (diffRatings[userItem] + userRating) * freq
                # 分母
                frequencies[diffItem] += freq

    recommendations = [(k, v / frequencies[k]) for (k, v) in recommendations.items()]
    # 排序并返回
    recommendations.sort(key=lambda artistTuple: artistTuple[1], reverse=True)
    return recommendations

>>> r.slopeOneRecommendations(users2['Ben'])
[('Whitney Houston', 3.375)]
```

MovieLens数据集

让我们在另一个数据集上尝试一下Slope One算法。MovieLens数据集是由明尼苏达州大学的GroupLens研究项目收集的，是用户对电影的评分。这个数据集可以在www.grouplens.org下载，有三种大小，这里我使用的是最小的那个，包含了943位用户对1682部电影的评价，约10万条记录。我们一起来测试一下：

```
>>> r = recommender(0)
>>> r.loadMovieLens('/Users/raz/Downloads/ml-100k/')
102625
>>> r.computeDeviations() # 大约需要50秒
>>> r.slopeOneRecommendations(r.data['25'])
[('Aiqing wansui (1994)', 5.674418604651163), ('Boys, Les (1997)', 5.523076923076923), ..
```

作业

1. 看看Slope One的推荐结果是否靠谱：对数据集中的10部电影进行评分，得到的推荐结果是否是你喜欢的电影呢？
2. 实现修正的余弦相似度算法，比较一下两者的运算效率。
3. （较难）我的笔记本电脑有8G内存，在尝试用Slope One计算图书漂流站数据集时报内存溢出了。那个数据集中有27万本书，因此需要保存超过7300万条记录的Python字典。这个字典的数据是否很稀疏呢？修改算法，让它能够处理更多数据吧。



祝贺大家学完第三章了

第四章：分类

原文：http://guidetodatamining.com/chapter-4

在上几章中我们使用用户对物品的评价来进行推荐，这一章我们将使用物品本身的特征来进行推荐。这也是潘多拉音乐站所使用的方法。

内容：

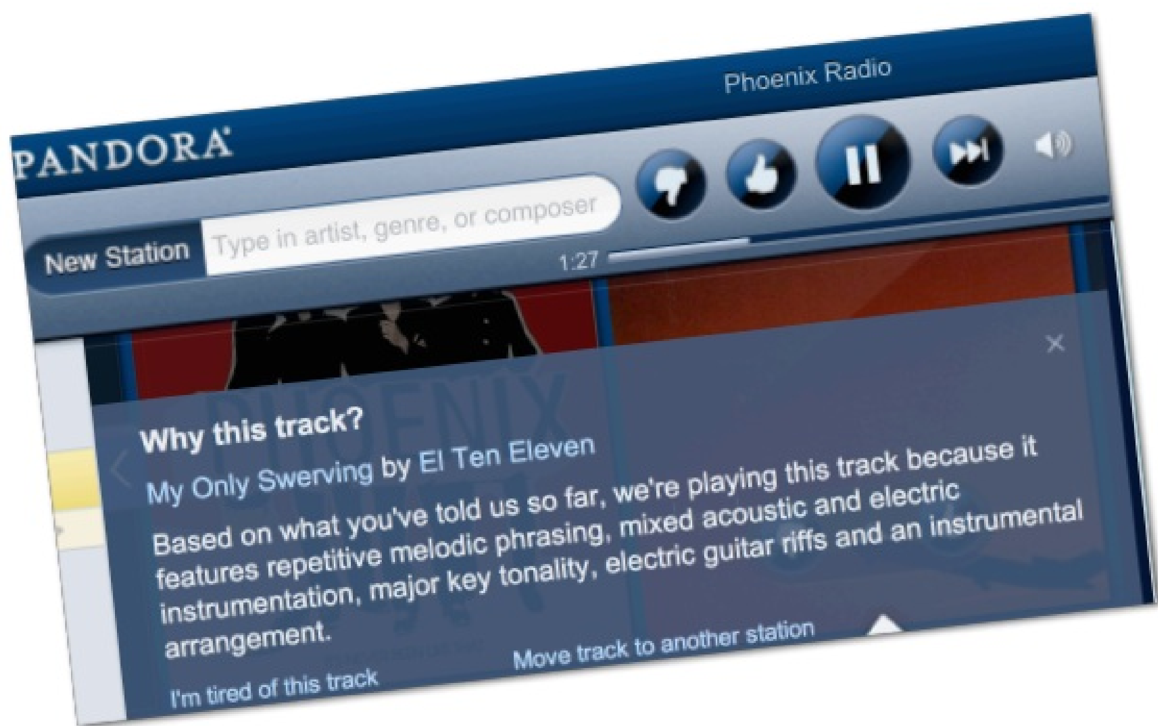
- 潘多拉推荐系统简介
- 特征值选择的重要性
- 示例：音乐特征值和邻域算法
- 数据标准化
- 修正的标准分数
- Python代码：音乐，特征，以及简单的邻域算法实现
- 一个和体育相关的示例
- 特征值抽取方式一览

根据物品特征进行分类

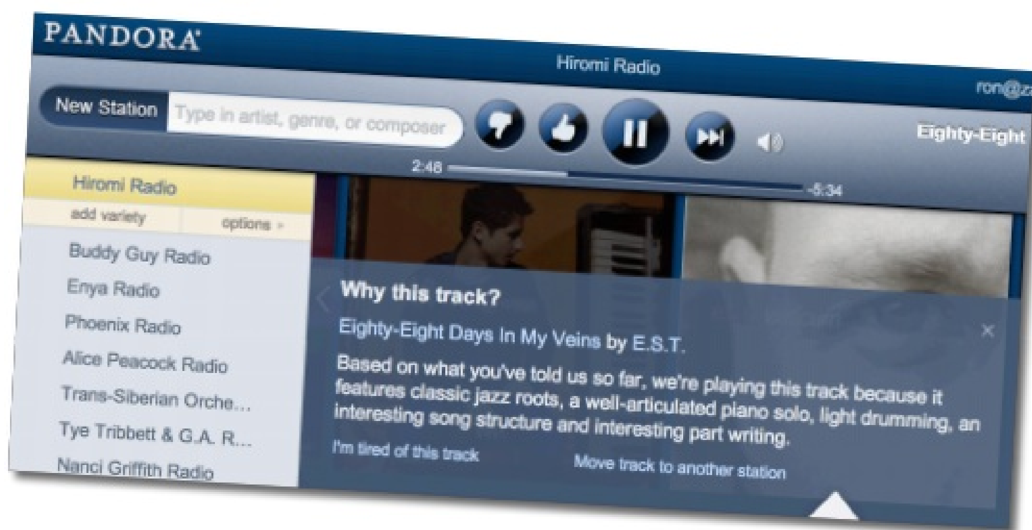
前几章我们讨论了如何使用协同过滤来进行推荐，由于使用的是用户产生的各种数据，因此又称为社会化过滤算法。比如你购买了Phoenix专辑，我们网站上其他购买过这张专辑的用户还会去购买Vampire的专辑，因此会把它推荐给你；我在Netflix上观看了Doctor Who，网站会向我推荐Quantum Leap，用的是同样的原理。我们同时也讨论了协同过滤会遇到的种种问题，包括数据的稀疏性和算法的可扩展性。此外，协同过滤算法倾向于推荐那些已经很流行的物品。试想一个极端的例子：一个新乐队发布了专辑，这张专辑还没有被任何用户评价或购买过，那它将永远不会出现在推荐列表中。

> 这类推荐系统会让流行的物品更为流行，冷门的物品更无人问津。
>
> -- Daniel Fleder & Kartik Hosanagar 2009 《推荐系统对商品分类的影响》

这一章我们来看另一种推荐方法。以潘多拉音乐站举例，在这个站点上你可以设立各种音乐频道，只需为这个频道添加一个歌手，潘多拉就会播放和这个歌手风格相类似的歌曲。比如我添加了Phoenix乐队，潘多拉便会播放El Ten Eleven的歌曲。它并没有使用协同过滤，而是通过计算得到这两个歌手的音乐风格是相似的。其实在播放界面上可以看到推荐理由：



“根据你目前告知的信息，我们播放的这首歌曲有着相似的旋律，使用了声响和电音的组合，即兴的吉他伴奏。”在我的Hiromi音乐站上，潘多拉会播放E.S.T.的歌曲，因为“它有着古典爵士乐风，一段高水准的钢琴独奏，轻盈的打击乐，以及有趣的歌曲结构。”



潘多拉网站的推荐系统是基于一个名为音乐基因的项目。他们雇佣了专业的音乐家对歌曲进行分类（提取它们的“基因”）。这些音乐家会接受超过150小时的训练，之后便可用20到30分钟的时间来分析一首歌曲。这些乐曲特征是很专业的：

El Ten Eleven		My Only Swerving	
Beats per Minute:	110	major tonality:	5
swinging 16ths:	0	electric guitar riffs:	5
well articulated piano solo:	2	repetitive melodic phrasing:	4
block chords:	3	drumming:	3
acoustic instrumentation:	5	electric instrumentation:	4

这些专家要甄别400多种特征，平均每个月会有15000首新歌曲，因此这是一项非常消耗人力的工程。

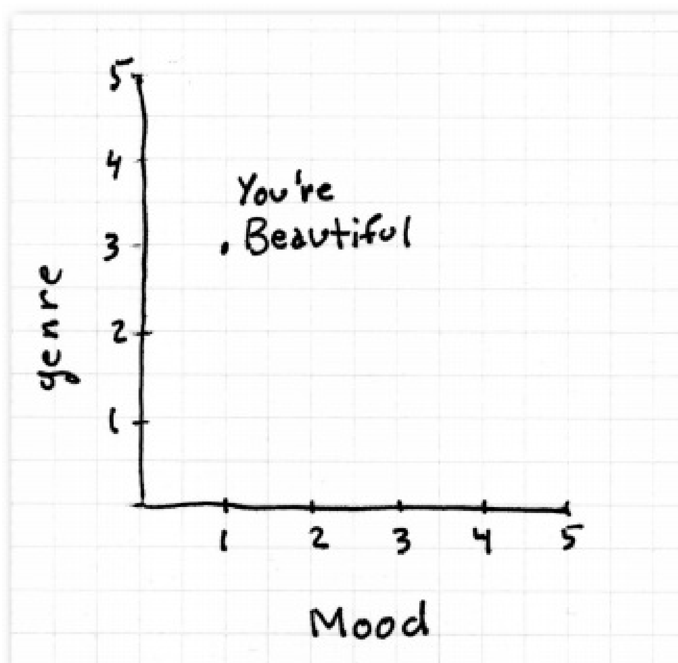
> 注意：潘多拉的音乐基因项目是商业机密，我不曾了解它的任何信息。下文讲述的是如何构造一个类似的系统。

特征值选取的重要性

假设潘多拉会用曲风和情绪作为歌曲特征，分值如下：

- 曲风：乡村1分，爵士2分，摇滚3分，圣歌4分，饶舌5分
- 情绪：悲伤的1分，欢快的2分，热情的3分，愤怒的4分，不确定的5分

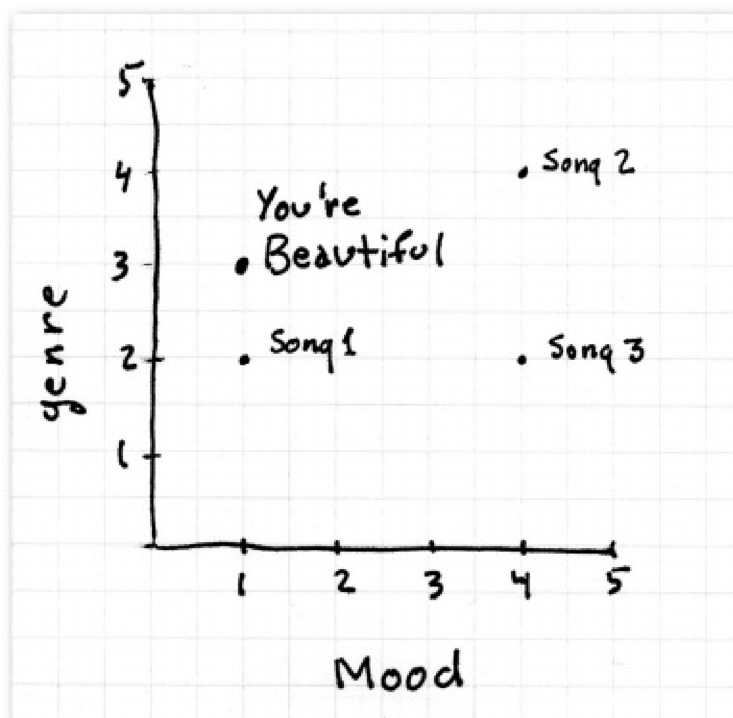
比如James Blunt的那首You're Beautiful是悲伤的摇滚乐，用图表来展示它的位置便是：



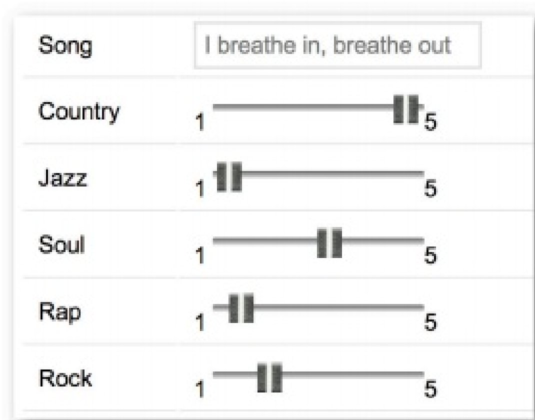
比如一个叫Tex的用户喜欢You're Beautiful这首歌，我们想要为他推荐歌曲。



我们的歌曲库中有另外三首歌：歌曲1是悲伤的爵士乐；歌曲2是愤怒的圣歌；歌曲3是愤怒的摇滚乐。你会推荐哪一首？



图中歌曲1看起来是最相近的。也许你已经看出了这种算法中的不足，因为不管用何种计算距离的公式，爵士乐和摇滚乐是相近的，悲伤的乐曲和快乐的乐曲是相近的等等。即使调整了分值的分配，也不能解决问题。这就是没有选取好特征值的例子。不过解决的方法也很简单，我们将每种歌曲类型拆分成单独的特征，并对此进行打分：



“乡村音乐”一栏的1分表示完全不是这个乐曲风格，5分则表示很相符。这样一来，评分值就显得有意义了。如果一首歌的“乡村音乐”特征是4分，另一首是5分，那我们可以认为它们是相似的歌曲。

其实这就是潘多拉所使用的特征抽取方法。每个特征都是1到5分的尺度，0.5分为一档。特征会被分到不同的大类中。通过这种方式，潘多拉将每首歌曲都抽象成一个包含400个数值元素的向量，并结合我们之前学过的距离计算公式进行推荐。

一个简单的示例

我们先来构建一个数据集，我选取了以下这些特征（可能比较随意），使用5分制来评分（0.5分一档）：

- 使用钢琴的程度（Piano）：1分表示没有使用钢琴，5分表示整首歌曲由钢琴曲贯穿；
- 使用美声的程度（Vocals）：标准同上
- 节奏（Driving beat）：整首歌曲是否有强烈的节奏感
- 蓝调（Blues infl.）
- 电音吉他（Dirty elec. Guitar）
- 幕后和声（Backup vocals）
- 饶舌（Rap infl.）

使用以上标准对一些歌曲进行评分：

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.
Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1
Phoenix/ Lisztomania	2	5	5	3	2	1	1
Heartless Bastards / Out at Sea	1	5	4	2	4	1	1
Todd Snider/ Don't Tempt Me	4	5	4	4	1	5	1
The Black Keys/ Magic Potion	1	4	5	3.5	5	1	1
Glee Cast/ Jessie's Girl	1	5	3.5	3	4	5	1
Black Eyed Peas/ Rock that Body	2	5	5	1	2	2	4
La Roux/ Bulletproof	5	5	4	2	1	1	1
Mike Posner/ Cooler than me	2.5	4	4	1	1	1	1
Lady Gaga/ Alejandro	1	5	3	2	1	2	1

然后我们便可以使用距离计算公式了，比如要计算Dr. Dog的Fate歌曲和Phoenix的Lisztomania之间的曼哈顿距离：

Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1
Phoenix/ Lisztomania	2	5	5	3	2	1	1
Distance	**0.5**	**1**	**1.5**	**0**	**3**	**3**	**0**

相加得到两首歌曲的曼哈顿距离为9。

使用Python实现推荐逻辑

回忆一下，我们在协同过滤中使用的用户评价数据是这样的：

```
users = {"Angelica": {"Blues Traveler": 3.5, "Broken Bells": 2.0, "Norah Jones": 4.5, "Ph
         "Bill":{"Blues Traveler": 2.0, "Broken Bells": 3.5, "Deadmau5": 4.0, "Phoenix":
```

我们将上文中的歌曲特征数据也用类似的格式储存起来：

```
music = {"Dr Dog/Fate": {"piano": 2.5, "vocals": 4, "beat": 3.5, "blues": 3, "guitar": 5,
         "Phoenix/Lisztomania": {"piano": 2, "vocals": 5, "beat": 5, "blues": 3, "guitar"
         "Heartless Bastards/Out at Sea": {"piano": 1, "vocals": 5, "beat": 4, "blues": 2
         "Todd Snider/Don't Tempt Me": {"piano": 4, "vocals": 5, "beat": 4, "blues": 4, "
         "The Black Keys/Magic Potion": {"piano": 1, "vocals": 4, "beat": 5, "blues": 3.5
         "Glee Cast/Jessie's Girl": {"piano": 1, "vocals": 5, "beat": 3.5, "blues": 3, "g
         "La Roux/Bulletproof": {"piano": 5, "vocals": 5, "beat": 4, "blues": 2, "guitar"
         "Mike Posner": {"piano": 2.5, "vocals": 4, "beat": 4, "blues": 1, "guitar": 1, "
         "Black Eyed Peas/Rock That Body": {"piano": 2, "vocals": 5, "beat": 5, "blues":
         "Lady Gaga/Alejandro": {"piano": 1, "vocals": 5, "beat": 3, "blues": 2, "guitar"
```

假设我有一个朋友喜欢Black Keys Magic Potion，我便可根据曼哈顿距离来进行推荐：

```
>>> computeNearestNeighbor('The Black Keys/Magic Potion', music)
[(4.5, 'Heartless Bastards/Out at Sea'), (5.5, 'Phoenix/Lisztomania'), (6.5, 'Dr Dog/Fate
```

这里我推荐的是Heartless Bastard的Out as Sea，还是很合乎逻辑的。当然，由于我们的数据集比较小，特征和歌曲都不够丰富，因此有些推荐结果并不太好。

这段代码可以点此浏览。

如何显示“推荐理由”？

潘多拉在推荐歌曲时会显示推荐理由，我们也可以做到这一点。比如在上面的例子中，我们可以将Magic Potion和Out at Sea的音乐特征做一个比较，找出高度相符的点：

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.
Black Keys Magic Potion	1	5	4	2	4	1	1
Heartless Bastards Out at Sea	1	4	5	3.5	5	1	1
difference	0	1	1	1.5	1	0	0

可以看到，两首歌曲最相似的地方是钢琴、和声、以及饶舌，这些特征的差异都是0。但是，这些特征的评分都很低，我们不能告诉用户“因为这首歌曲没有钢琴伴奏，所以我们推荐给你”。因此，我们需要使用那些相似的且评分较高的特征。



我们推荐歌曲是因为它有着强烈的节奏感，美声片段，以及电音吉他的演奏。

评分标准的问题

假如我想增加一种音乐特征——每分钟的鼓点数（bpm），用来判断这是一首快歌还是慢歌。以下是扩充后的数据集：

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.	bpm
Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1	140
Phoenix/ Lisztomania	2	5	5	3	2	1	1	110
Heartless Bastards / Out at Sea	1	5	4	2	4	1	1	130
The Black Keys/ Magic Potion	1	4	5	3.5	5	1	1	88
Glee Cast/ Jessie's Girl	1	5	3.5	3	4	5	1	120
Bad Plus/ Smells like Teen Spirit	5	1	2	1	1	1	1	90

没有bpm时，Magic Potion和Out at Sea距离最近，和Smells Like Teen Spirit距离最远。但引入bpm后，我们的结果就乱套了，因为bpm基本上就决定了两首歌的距离。现在Bad Plus和The Black Keys距离最近就是因为bpm数据相近。

再举个有趣的例子。在婚恋网站上，我通过用户的年龄和收入来进行匹配：

gals

name	age	salary
Yun L	35	75,000
Allie C	52	55,000
Daniela C	27	45,000
Rita A	37	115,000

guys

name	age	salary
Brian A	53	70,000
Abdullah K	25	105,000
David A	35	69,000
Michael W	48	43,000

这样一来，年龄的最大差异是28，而薪资的最大差异则是72,000。因为差距悬殊，薪水的高低基本决定了匹配程度。如果单单目测，我们会将David推荐给Yun，因为他们年龄相近，工资也差不多。但如果使用距离计算公式，那么53岁的Brian就会被匹配给Yun，这就不太妙了。



事实上，评分标准不一是所有推荐系统的大敌！

标准化

不用担心，我们可以使用标准化。



要让数据变得可用我们可以对其进行标准化，最常用的方法是将所有数据都转化为0到1之间的值。

拿上面的薪酬数据举例，最大值115,000和最小值43,000相差72,000，要让所有值落到0到1之间，可以将每个值减去最小值，并除以范围（72,000）。所以，Yun标准化之后的薪水是：

$$(75{,}000 - 43{,}000) / 72{,}000 = 0.444$$

对一些数据集，这种简单的方法效果是不错的。

gals		
name	salary	normalized salary
Yun L	75,000	0.444
Allie C	55,000	0.167
Daniela C	45,000	0.028
Rita A	115,000	1.0

如果你学过统计学，会知道还有其他的标准化方法。比如说标准分（z-score）——分值偏离均值的程度：

$$\frac{\text{(each value)} - \text{(mean)}}{\text{(standard deviation)}} = \text{Standard Score}$$



标准差的计算公式是：

$$sd = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{card(x)}}$$

card(x)表示集合x中的元素个数。

> 如果你对统计学有兴趣，可以读一读《漫话统计学》。

我们用上文中交友网站的数据举例。所有人薪水的总和是577,000，一共有8人，所以均值为72,125。代入标准差的计算公式：

Yun's salary　　Allie's salary　　Daniela's salary　　etc.

$$\sqrt{\frac{(75{,}000 - 72{,}125)^2 + (55{,}000 - 72{,}125)^2 + (45{,}000 - 72{,}125)^2 + \ldots}{8}}$$

$$= \sqrt{\frac{8{,}265{,}625 + 293{,}265{,}625 + 735{,}765{,}625 + \ldots}{8}} = \sqrt{602{,}395{,}375}$$

$$= 24{,}543.01$$

那Yun的标准分则是：

$$\frac{75000-72125}{24543.01}=\frac{2875}{24543.01}=0.117$$

练习题：计算**Allie**、**Daniela**、**Rita**的标准分

name	salary	Standard Score
Yun L	75,000	0.117
Allie C	55,000	-0.698
Daniela C	45,000	-1.105
Rita A	115,000	1.747

Allie:
(55,000 - 72,125) / 24,543.01
= -0.698

Daniela:
(45,000 - 72,125) / 24,543.01
= -1.105

Rita:
(115,000 - 72,125) / 24,543.01
= 1.747

标准分带来的问题

标准分的问题在于它会受异常值的影响。比如说一家公司有100名员工，普通员工每小时赚10美元，而CEO一年能赚600万，那全公司的平均时薪为：

(100 * $10 + 6,000,000 / (40 * 52)) / 101

= (1000 + 2885) / 101 = $38/hr.

结果是每小时38美元，看起来很美好，但其实并不真实。鉴于这个原因，标准分的计算公式会稍作变化。

修正的标准分

计算方法：将标准分公式中的均值改为中位数，将标准差改为绝对偏差。以下是绝对偏差的计算公式：



$$asd = \frac{1}{card(x)} \sum_i |x_i - \mu|$$

中位数指的是将所有数据进行排序，取中间的那个值。如果数据量是偶数，则取中间两个数值的均值。

下面就让我们试试吧。首先将所有人按薪水排序，找到中位数，然后计算绝对偏差：

Name	Salary
Michael W	43,000
Daniela C	45,000
Allie C	55,000
David A	69,000
Brian A	70,000
Yun L	75,000
Abdullah K	105,000
Rita A	115,000

$$median = \frac{(69{,}000 + 70{,}000)}{2} = 69{,}500$$

$$asd = \frac{1}{card(x)} \sum_i |x_i - \mu|$$

$$asd = \frac{1}{8}(|43{,}000 - 69{,}500| + |45{,}000 - 69{,}500| + |55{,}000 - 69{,}500)| + \ldots)$$

$$= \frac{1}{8}(26{,}500 + 24{,}500 + 14{,}500 + 500 + \ldots)$$

$$= \frac{1}{8}(153{,}000) = 19{,}125$$

最后，我们便可以计算得出Yun的修正标准分：

Modified Standard Score:

$$\frac{\text{(each value)} - \text{(median)}}{\text{(absolute standard deviation)}}$$

$$mss = \frac{(75{,}000 - 69{,}500)}{19{,}125} = \frac{5{,}500}{19{,}125} = 0.2876$$

是否需要标准化？

当物品的特征数值尺度不一时，就有必要进行标准化。比如上文中音乐特征里大部分是1到5分，鼓点数却是60到180；交友网站中薪水和年龄这两个尺度也有很大差别。

再比如我想在新墨西哥圣达菲买一处宅子，下表是一些选择：

asking price	bedrooms	bathrooms	sq. ft.
$1,045,000	2	2.0	1,860
$1,895,000	3	4.0	2,907
$3,300,000	6	7.0	10,180
$6,800,000	5	6.0	8,653
$2,250,000	3	2.0	1,030

可以看到，价格的范围是最广的，在计算距离时会起到决定性作用；同样，有两间卧室和有二十间卧室，在距离的影响下作用也会很小。

需要进行标准化的情形：

1. 我们需要通过物品特性来计算距离；
2. 不同特性之间的尺度相差很大。

但对于那种“赞一下”、“踩一脚”的评分数据，就没有必要做标准化了：

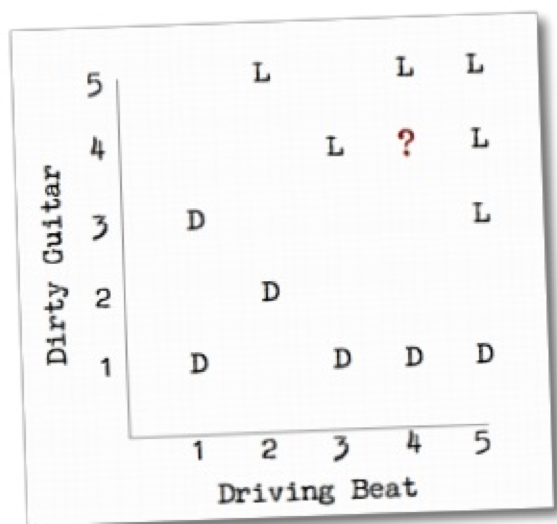
Bill = {0, 0, 0, 1, 1, 1, 1, 0, 1, 0 ... }

在潘多拉的例子中，如果所有的音乐特征都是在1到5分之间浮动的，是否还需要标准化呢？虽然即使做了也不会影响计算结果，但是任何运算都是有性能消耗的，这时我们可以通过比较两种方式的性能和效果来做进一步选择。在下文中，我们会看到标准化反而会降低结果正确性的示例。

回到潘多拉

在潘多拉网站的示例中，我们用一个特征向量来表示一首歌曲，用以计算歌曲的相似度。潘多拉网站同样允许用户对歌曲“赞”和“踩”，那我们要如何利用这些数据呢？

假设我们的歌曲有两个特征，重金属吉他（Dirty Guitar）和强烈的节奏感（Driving Beat），两种特征都在1到5分之间。一位用户对5首歌曲做了“赞”的操作（图中的L），另外五首则“踩”了一下（图中的D）：



图中多了一个问号所表示的歌曲，你觉得用户会喜欢它还是不喜欢呢？想必你也猜到了，因为这个问号离用户喜欢的歌曲距离较近。这一章接下来的篇幅都会用来讲述这种计算方法。最明显的方式是找到问号歌曲最邻近的歌曲，因为它们之间相似度比较高，再根据用户是否喜欢这些邻近歌曲来判断他对问号歌曲的喜好。

使用Python实现最邻近分类算法

我们仍使用上文中的歌曲示例，用7个特征来标识10首歌曲：

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.
Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1
Phoenix/ Lisztomania	2	5	5	3	2	1	1
Heartless Bastards / Out at Sea	1	5	4	2	4	1	1
Todd Snider/ Don't Tempt Me	4	5	4	4	1	5	1
The Black Keys/ Magic Potion	1	4	5	3.5	5	1	1
Glee Cast/ Jessie's Girl	1	5	3.5	3	4	5	1
Black Eyed Peas/ Rock that Body	2	5	5	1	2	2	4
La Roux/ Bulletproof	5	5	4	2	1	1	1
Mike Posner/ Cooler than me	2.5	4	4	1	1	1	1
Lady Gaga/ Alejandro	1	5	3	2	1	2	1

使用Python代码来表示这些数据：

```
music = {"Dr Dog/Fate": {"piano": 2.5, "vocals": 4, "beat": 3.5, "blues": 3, "guitar": 5,
         "Phoenix/Lisztomania": {"piano": 2, "vocals": 5, "beat": 5, "blues": 3, "guitar"
         "Heartless Bastards/Out at Sea": {"piano": 1, "vocals": 5, "beat": 4, "blues": 2
         "Todd Snider/Don't Tempt Me": {"piano": 4, "vocals": 5, "beat": 4, "blues": 4, "
         "The Black Keys/Magic Potion": {"piano": 1, "vocals": 4, "beat": 5, "blues": 3.5
         "Glee Cast/Jessie's Girl": {"piano": 1, "vocals": 5, "beat": 3.5, "blues": 3, "g
         "La Roux/Bulletproof": {"piano": 5, "vocals": 5, "beat": 4, "blues": 2, "guitar"
         "Mike Posner": {"piano": 2.5, "vocals": 4, "beat": 4, "blues": 1, "guitar": 1, "
         "Black Eyed Peas/Rock That Body": {"piano": 2, "vocals": 5, "beat": 5, "blues":
         "Lady Gaga/Alejandro": {"piano": 1, "vocals": 5, "beat": 3, "blues": 2, "guitar"
```

这样做虽然可行，但却比较繁琐，piano、vocals这样的键名需要重复很多次。我们可以将其简化为向量，即Python中的数组类型：

```
#
# 物品向量中的特征依次为：piano, vocals, beat, blues, guitar, backup vocals, rap
#
items = {"Dr Dog/Fate": [2.5, 4, 3.5, 3, 5, 4, 1],
         "Phoenix/Lisztomania": [2, 5, 5, 3, 2, 1, 1],
         "Heartless Bastards/Out": [1, 5, 4, 2, 4, 1, 1],
         "Todd Snider/Don't Tempt Me": [4, 5, 4, 4, 1, 5, 1],
         "The Black Keys/Magic Potion": [1, 4, 5, 3.5, 5, 1, 1],
         "Glee Cast/Jessie's Girl": [1, 5, 3.5, 3, 4, 5, 1],
         "La Roux/Bulletproof": [5, 5, 4, 2, 1, 1, 1],
         "Mike Posner": [2.5, 4, 4, 1, 1, 1, 1],
         "Black Eyed Peas/Rock That Body": [2, 5, 5, 1, 2, 2, 4],
         "Lady Gaga/Alejandro": [1, 5, 3, 2, 1, 2, 1]}
```

在线性代数中，向量（vector）指的是具有大小和方向的几何对象。向量支持多种运算，包括相加、相减、以及数乘等。



在数据挖掘中，向量则可简单认为是物品的一组特征，比如上文中音乐歌曲的特征。做文本挖掘时，会将一篇文章也用向量来表示——每个元素的位置表示一个特定的单词，这个位置上的值表示单词出现的次数。



此外，用“向量”一词比用“物品的一组特征”要来得专业。

当我们用这种方式定义特征后，就可以运用线性代数中的向量运算法则了。

接下来我还需要将用户“赞”和“踩”的数据也用Python代码表示出来。由于用户并不会对所有的歌曲都做这些操作，所以我用嵌套的字典来表示：

```
users = {"Angelica": {"Dr Dog/Fate": "L",
                      "Phoenix/Lisztomania": "L",
                      "Heartless Bastards/Out at Sea": "D",
                      "Todd Snider/Don't Tempt Me": "D",
                      "The Black Keys/Magic Potion": "D",
                      "Glee Cast/Jessie's Girl": "L",
                      "La Roux/Bulletproof": "D",
                      "Mike Posner": "D",
                      "Black Eyed Peas/Rock That Body": "D",
                      "Lady Gaga/Alejandro": "L"},
         "Bill": {"Dr Dog/Fate": "L",
                  "Phoenix/Lisztomania": "L",
                  "Heartless Bastards/Out at Sea": "L",
                  "Todd Snider/Don't Tempt Me": "D",
                  "The Black Keys/Magic Potion": "L",
                  "Glee Cast/Jessie's Girl": "D",
                  "La Roux/Bulletproof": "D",
                  "Mike Posner": "D",
                  "Black Eyed Peas/Rock That Body": "D",
                  "Lady Gaga/Alejandro": "D"}}
```

这里使用L和D两个字母来表示喜欢和不喜欢，当然你也可以用其他方式，比如0和1等。

对于新的向量格式，我们需要对曼哈顿距离函数和邻近物品函数做一些调整：

```
def manhattan(vector1, vector2):
    distance = 0
    total = 0
    n = len(vector1)
    for i in range(n):
        distance += abs(vector1[i] - vector2[i])
    return distance

def computeNearestNeighbor(itemName, itemVector, items):
    """按照距离排序，返回邻近物品列表"""
    distances = []
    for otherItem in items:
        if otherItem != itemName:
            distance = manhattan(itemVector, items[otherItem])
            distances.append((distance, otherItem))
    # 最近的排在前面
    distances.sort()
    return distances
```

最后，我需要建立一个分类函数，用来预测用户对一个新物品的喜好，如：

```
"Chris Cagle/I Breathe In. I Breathe Out" [1, 5, 2.5, 1, 1, 5, 1]
```

这个函数会先计算出与这个物品距离最近的物品，然后找到用户对这个最近物品的评价，以此作为新物品的预测值。下面是一个最简单的分类函数：

```
def classify(user, itemName, itemVector):
    nearest = computeNearestNeighbor(itemName, itemVector, items)[0][1]
    rating = users[user][nearest]
    return rating
`
```

让我们试用一下：

```
>>> classify('Angelica', 'Chris Cagle/I Breathe In. I Breathe Out', [1, 5, 2.5, 1, 1, 5,
'L'
```

我们认为她会喜欢这首歌曲！为什么呢？

```
>>> computeNearestNeighbor('Chris Cagle/I Breathe In. I Breathe Out', [1, 5, 2.5, 1, 1, 5
[(4.5, 'Lady Gaga/Alejandro'), (6.0, "Glee Cast/Jessie's Girl"), (7.5, "Todd Snider/Don't
```

可以看到，距离I Breathe In最近的歌曲是Alejandro，并且Angelica是喜欢这首歌曲的，所以我们预测她也会喜欢I Breathe In。

其实我们做的是一个分类器，将歌曲分为了用户喜欢和不喜欢两个类别。

号外，号外！我们编写了一个分类器！



分类器是指通过物品特征来判断它应该属于哪个组或类别的程序！

分类器程序会基于一组已经做过分类的物品进行学习，从而判断新物品的所属类别。在上面的例子中，我们知道Angelica喜欢和不喜欢的歌曲，然后据此判断她是否会喜欢Chris Cagle的歌。

1. 在Angelica评价过的歌曲中找到距离Chris Cagle最近的歌曲，即Laydy Gaga的Alejandro；
2. 由于Angelica是喜欢Alejandro这首歌的，所以我们预测她也会喜欢Chris Cagle的Breathe In, Breathe Out。

分类器的应用范围很广，以下是一些示例：

推特情感分类

很多人在对推特中的文字消息进行情感分类（积极的、消极的），可以有很多用途，如Axe发布了一款新的腋下除臭剂，通过推文就能知道用户是否满意。这里用到的物品特征是文字信息。

人脸识别

现在有些手机应用可以识别出照片里你的朋友们，这项技术也可用于监控录像中的人脸识别。不同的识别技术细节可能不同，但都会用到诸如五官的大小和相对距离等信息。

政治拉票

通过将目标选民分为“爱凑热闹”、“很有主见”、“家庭为重”等类型，来进行有针对性的拉票活动。

市场细分

这和上个例子有点像，与其花费巨额广告费向不可能购买维加斯公寓的人进行宣传，不如从人群中识别出潜在客户，缩小宣传范围。最好能再对目标群体进行细分，进一步定制广告内容。

个人健康助理

如今人们越来越关注自身，我们可以购买到像Nike健身手环这样的产品，而Intel等公司也在研制一种智能家居，可以在你行走时称出你的重量，记录你的行动轨迹，并给出健康提示。有些专家还预言未来我们会穿戴各种便携式设备，收集我们的生活信息，并加以分类。

其他

- 识别恐怖分子
- 来信分类（重要的、一般的、垃圾邮件）
- 预测医疗费用
- 识别金融诈骗

她是从事什么运动的？

让我们来为之后的几章做一个预热，先看一个较为简单的例子——根据女运动员的身高和体重来判断她们是从事什么运动项目的。下表是原始数据：

Name	Sport	Age	Height	Weight
Asuka Teramoto	Gymnastics	16	54	66
Brittainey Raven	Basketball	22	72	162
Chen Nan	Basketball	30	78	204
Gabby Douglas	Gymnastics	16	49	90
Helalia Johannes	Track	32	65	99
Irina Miketenko	Track	40	63	106
Jennifer Lacy	Basketball	27	75	175
Kara Goucher	Track	34	67	123
Linlin Deng	Gymnastics	16	54	68
Nakia Sanford	Basketball	34	76	200
Nikki Blue	Basketball	26	68	163
Qiushuang Huang	Gymnastics	20	61	95
Rebecca Tunney	Gymnastics	16	58	77
Rene Kalmer	Track	32	70	108
Shanna Crossley	Basketball	26	70	155
Shavonte Zellous	Basketball	24	70	155
Tatyana Petrova	Track	29	63	108
Tiki Gelana	Track	25	65	106
Valeria Straneo	Track	36	66	97
Viktoria Komova	Gymnastics	17	61	76

这里列出的是2008和2012奥运会上排名靠前的二十位女运动员。篮球运动员参加了WNBA；田径运动员则完成了2012年奥运会的马拉松赛。虽然数据量很小，但我们仍可以对其应用一些数据挖掘算法。

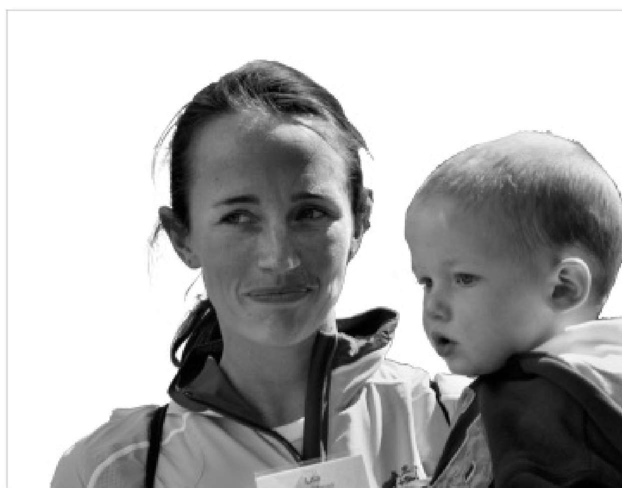
你可以看到上表中列出了运动员的年龄，光凭这一信息就能进行一些预测了。比如，以下运动员会是哪个项目的呢？



Candace Parker; Age 26



McKayla Maroney; Age 16



Lisa Jane Weightman; Age 34



Olivera Jevtić: Age 35

答案

Candace Parker是篮球运动员，McKayla Maroney是美国女子体操队的一员，Olivera Jevtic是塞尔维亚的一名长跑运动员，Lisa Jane Weightman则是澳大利亚的长跑运动员。

看，我们刚刚就进行了一次分类——通过运动员的年龄特征来识别她们参与的体育项目。

头脑风暴

假设我想通过运动员的身高和体重来预测她所从事的运动，数据集只有两人：Nakia Sanford是篮球运动员，身高6尺4寸（76英寸，1.93米），体重200磅（90公斤）；Sarah Beale是橄榄球运动员，身高5尺10寸（70英寸，1.78米），体重190磅（86公斤）。我想知道Catherine Spencer是从事哪项运动的，她的身高是5尺10寸，重200磅，如何预测呢？

如果你认为她是橄榄球运动员，那么你猜对了。但是，如果用曼哈顿距离来进行计算，Catherine和Nakia的距离是6，和Sarah的距离是10，那应该预测她是篮球运动员才对。我们之前是否学过一个方法，能让距离计算更为准确呢？

没错，就是修正的标准分！

测试数据

下表是我们需要进行预测的运动员列表，一起来做分类器吧！

Name	Sport	Height	Weight
Crystal Langhorne		74	190
Li Shanshan		64	101
Kerri Strug		57	87
Jaycie Phelps		60	97
Kelly Miller		70	140
Zhu Xiaolin		67	123
Lindsay Whalen		69	169
Koko Tsurumi		55	75
Paula Radcliffe		68	120
Erin Thorn		69	144

Python编码

这次我们不将数据直接写在Python代码中，而是放到两个文本文件里：athletesTrainingSet.txt和athletesTestSet.txt。我会使用第一个文件中的数据来训练分类器，然后使用测试文件里的数据来进行评价。

文件格式大致如下：

Asuka Teramoto	Gymnastics	54	66
Brittainey Raven	Basketball	72	162
Chen Nan	Basketball	78	204
Gabby Douglas	Gymnastics	49	90

文件中的每一行是一条完整的记录，字段使用制表符分隔。我要使用运动员的身高体重数据来预测她所从事的运动项目，也就是用第三、四列的数据来预测第二列的数据。运动员的姓名不会使用到，我们既不能通过运动员的姓名得知她参与的项目，也不会通过身高体重来预测运动员的姓名。



> 你好，你有五英尺高，150磅重，莫非你的名字是Clara Coleman？

当然，名字也有它的用处，我们可以用它来解释分类器的预测结果："我们认为Amelia Pond是一名体操运动员，因为她的身高体重和另一名体操运动员Gabby Douglas很接近。"

为了让我们的Python代码更具一般性，并不只适用于这一种数据集，我会为每一列数据增加一个列名，如：

comment	class	num	num
Asuka Teramoto	Gymnastics	54	66
Brittainey Raven	Basketball	72	162

所有被标记为comment的列都会被分类器忽略；标记为class的列表示物品所属分类；不定个数的num列则表示物品的特征。

头脑风暴

我们在Python中应该如何表示这些数据呢？以下是一些可能性：

```
# 1
{'Asuka Termoto': ('Gymnastics', [54, 66]),
 'Brittainey Raven': ('Basketball', [72, 162]), ...
```

这种方式使用了运动员的姓名作为键，而我们说过分类器程序根本不会使用到姓名，所以不合理。

```
# 2
[['Asuka Termoto', 'Gymnastics', 54, 66],
 ['Brittainey Raven', 'Basketball', 72, 162], ...
```

这种方式看起来不错，它直接反映了文件的格式。由于我们需要遍历文件的数据，所以使用列表类型（list）是合理的。

```
# 3
[('Gymnastics', [54, 66], ['Asuka Termoto']),
 ('Basketball', [72, 162], ['Brittainey Raven']), ...
```

这是我最认同的表示方式，因为它将不同类型的数据区别开来了，依次是分类、特征、备注。这里备注可能有多个，所以也用了一个列表来表示。以下是读取数据文件并转换成上述格式的函数：

```
class Classifier:

    def __init__(self, filename):

        self.medianAndDeviation = []

        # 读取文件
        f = open(filename)
        lines = f.readlines()
        f.close()
        self.format = lines[0].strip().split('\t')
        self.data = []
        for line in lines[1:]:
            fields = line.strip().split('\t')
            ignore = []
            vector = []
            for i in range(len(fields)):
                if self.format[i] == 'num':
                    vector.append(int(fields[i]))
                elif self.format[i] == 'comment':
                    ignore.append(fields[i])
                elif self.format[i] == 'class':
                    classification = fields[i]
            self.data.append((classification, vector, ignore))
```

动手实践

在计算修正的标准分之前，我们需要编写获取中位数和计算绝对偏差的函数，尝试实现这两个函数：

```
>>> heights = [54, 72, 78, 49, 65, 63, 75, 67, 54]
>>> median = classifier.getMedian(heights)
>>> median
65
>>> asd = classifier.getAbsoluteStandardDeviation(heights, median)
>>> asd
8.0
```

关于断言

通常我们会将一个大的算法拆分成几个小的组件，并为每个组件编写一些单元测试，从而确保它能正常工作。很多时候，我们会先写单元测试，再写正式的代码。在我提供的模板代码中已经编写了一些单元测试，摘录如下：

```
def unitTest():
    list1 = [54, 72, 78, 49, 65, 63, 75, 67, 54]
    classifier = Classifier('athletesTrainingSet.txt')
    m1 = classifier.getMedian(list1)
    assert(round(m1, 3) == 65)
    ...
    print("getMedian和getAbsoluteStandardDeviation均能正常工作")
```

你需要完成的geMedian函数的模板是：

```
def getMedian(self, alist):
    """返回中位数"""

    """请在此处编写代码"""
    return 0
```

这个模板函数返回的是0，你需要编写代码来返回列表的中位数。比如单元测试中我传入了以下列表：

```
[54, 72, 78, 49, 65, 63, 75, 67, 54]
```

assert（断言）表示函数的返回值应该是65。如果所有的单元测试都能通过，则报告以下信息：

```
getMedian和getAbsoluteStandardDeviation均能正常工作
```

否则，则抛出以下异常：

```
File "testMedianAndASD.py", line 78, in unitTest
    assert(round(m1, 3) == 65)
AssertError
```

断言在单元测试中是很常用的。

> 将大型代码拆分成一个个小的部分，并为每个部分编写单元测试，这一点是很重要的。如果没有单元测试，你将无法知道自己是否正确完成了所有任务，以及未来的某个修改是否会导致你的程序不可用。--- Peter Norvig



答案

```
def getMedian(self, alist):
    """返回中位数"""
    if alist == []:
        return []
    blist = sorted(alist)
    length = len(alist)
    if length % 2 == 1:
        # 列表有奇数个元素，返回中间的元素
        return blist[int(((length + 1) / 2) -  1)]
    else:
        # 列表有偶数个元素，返回中间两个元素的均值
        v1 = blist[int(length / 2)]
        v2 =blist[(int(length / 2) - 1)]
        return (v1 + v2) / 2.0

def getAbsoluteStandardDeviation(self, alist, median):
    """计算绝对偏差"""
    sum = 0
    for item in alist:
        sum += abs(item - median)
    return sum / len(alist)
```

可以看到，getMedian函数对列表进行了排序，由于数据量并不大，所以这种方式是可以接受的。如果要对代码进行优化，我们可以使用选择算法。

现在，我们已经将数据从athletesTrainingSet.txt读取出来，并保存为以下形式：

```
[('Gymnastics', [54, 66], ['Asuka Teramoto']),
 ('Basketball', [72, 162], ['Brittainey Raven']),
 ('Basketball', [78, 204], ['Chen Nan']),
 ('Gymnastics', [49, 90], ['Gabby Douglas']), ...
```

我们需要对向量中的数据进行标准化，变成以下结果：

```
[('Gymnastics', [-1.93277, -1.21842], ['Asuka Teramoto']),
 ('Basketball', [1.09243, 1.63447], ['Brittainey Raven']),
 ('Basketball', [2.10084, 2.88261], ['Chen Nan']),
 ('Gymnastics', [-2.7731, -0.50520],
 ('Track', [-0.08403, -0.23774], ['Helalia Johannes']),
 ('Track', [-0.42017, -0.02972], ['Irina Miketenko']), ...
```

在init方法中，添加标准化过程：

```
    # 获取向量的长度
    self.vlen = len(self.data[0][1])
    # 标准化
    for i in range(self.vlen):
        self.normalizeColumn(i)
```

在for循环中逐列进行标准化，即第一次会标准化身高，第二次标准化体重。

动手实践 下载normalizeColumnTemplate.py文件，编写normalizeColumn方法。

答案

```
def normalizeColumn(self, columnNumber):
    """标准化self.data中的第columnNumber列"""
    # 将该列的所有值提取到一个列表中
    col = [v[1][columnNumber] for v in self.data]
    median = self.getMedian(col)
    asd = self.getAbsoluteStandardDeviation(col, median)
    #print("Median: %f   ASD = %f" % (median, asd))
    self.medianAndDeviation.append((median, asd))
    for v in self.data:
        v[1][columnNumber] = (v[1][columnNumber] - median) / asd
```

可以看到，我将计算得到的中位数和绝对偏差保存在了medianAndDeviation变量中，因为我们会用它来标准化需要预测的向量。比如，我要预测Kelly Miller的运动项目，她身高5尺10寸（70英寸），重140磅，即原始向量为[70, 140]，需要先进行标准化。

我们计算得到的meanAndDeviation为：

```
[(65.5, 5.95), (107.0, 33.65)]
```

它表示向量中第一元素的中位数为65.5，绝对偏差为5.95；第二个元素的中位数为107.0，绝对偏差33.65。

现在我们就利用这组数据将[70, 140]进行标准化。第一个元素的标准分数是：

$$mss = \frac{x_i - \tilde{x}}{asd} = \frac{70 - 65.5}{5.95} = \frac{4.5}{5.95} = 0.7563$$

第二个元素为：

$$mss = \frac{x_i - \tilde{x}}{asd} = \frac{140 - 107}{33.65} = \frac{33}{33.65} = 0.98068$$

以下是实现它的Python代码：

```
def normalizeVector(self, v):
    """我们已保存了每列的中位数和绝对偏差，现用它来标准化向量v"""
    vector = list(v)
    for i in range(len(vector)):
        (median, asd) = self.medianAndDeviation[i]
        vector[i] = (vector[i] - median) / asd
    return vector
```

最后，我们要编写分类函数，用来预测运动员的项目：

```
classifier.classify([70, 140])
```

在我们的实现中，classify函数只是nearestNeighbor的一层包装：

```
def classify(self, itemVector):
    """预测itemVector的分类"""
    return self.nearestNeighbor(self.normalizeVector(itemVector))[1][0]
```

动手实践 实现nearestNeighbor函数。

答案

```
def manhattan(self, vector1, vector2):
    """计算曼哈顿距离"""
    return sum(map(lambda v1, v2: abs(v1 - v2), vector1, vector2))

def nearestNeighbor(self, itemVector):
    """返回itemVector的近邻"""
    return min([(self.manhattan(itemVector, item[1]), item)
                for item in self.data])
```

好了，我们用**200**多行代码实现了近邻分类器！



在完整的示例代码中，我提供了一个test函数，它可以对分类器程序的准确性做一个评价。比如用它来评价上面实现的分类器：

```
-        Track  Aly Raisman     Gymnastics    62    115
+   Basketball  Crystal Langhorne     Basketball    74    190
+   Basketball  Diana Taurasi     Basketball    72    163
...
+        Track  Xueqin Wang     Track    64    110
+        Track  Zhu Xiaolin     Track    67    123

80.00% correct
```

可以看到，这个分类器的准确率是80%。它对篮球运动员的预测很准确，但在预测田径和体操运动员时出现了4个失误。

鸢尾花数据集

我们可以用鸢尾花数据集做测试，这个数据集在数据挖掘领域是比较有名的。它是20世纪30年代Ronald Fisher对三种鸢尾花的50个样本做的测量数据（萼片和花瓣）。



> Ronald Fisher是一名伟大的科学家。他对统计学做出了革命性的改进，Richard Dawkins称他为“继达尔文后最伟大生物学家。”



鸢尾花数据集可以在这里找到，你可以测试你的算法，并问自己一些问题：标准化让结果更正确了吗？训练集中的数据量越多越好吗？用欧几里得距离来算会怎样？

记住 所有的学习过程都是在你自己的脑中进行的，你付出的努力越多，学到的也就越多。

鸢尾花数据集的格式如下，我们要预测的是Species这一列：



Sepal length	Sepal width	Petal Length	Petal Width	Species
5.1	3.5	1.4	0.2	I.setosa
4.9	3.0	1.4	0.2	I setosa

训练集中有120条数据，测试集中有30条，两者没有交集。

测试结果如何呢？

```
>>> test('irisTrainingSet.data', 'iristestSet.data')
93.33% correct
```

这又一次证明我们的分类算法是简单有效的。有趣的是，如果不对数据进行标准化，它的准确率将达到100%。这个现象我们会在后续的章节中讨论。

每加仑燃油可以跑多少公里？

最后，我们再来测试另一个广泛使用的数据集，卡内基梅隆大学统计的汽车燃油消耗和公里数数据。它在1983年的美国统计联合会展中使用过，大致格式如下：

mpg	cylinders	c.i.	HP	weight	secs. 0-60	make/model
30	4	68	49	1867	19.5	fiat 128
45	4	90	48	2085	21.7	vw rabbit (diesel)
20	8	307	130	3504	12	chevrolet chevelle malibu

这个数据集做过一些修改。我们要预测的是加仑燃油公里数（mpg），使用的数据包括汽缸数、排气量、马力、重量、加速度等。



数据集中有342条记录，50条测试记录，运行结果如下：

```
>>> test('mpgTrainingSet.txt', 'mpgTestSet.txt')
56.00% correct
```

如果不进行标准化，准确率将只有32%。



> 我们应该如何提高预测的准确率呢？改进分类算法？增加训练集？还是增加特征的数量？我们将在下一章揭晓！

番外篇：关于标准化

这一章我们讲解了标准化的重要性，即当不同特征的评分尺度不一致时，为了得到更准确的距离结果，就需要将这些特征进行标准化，使他们在同一个尺度内波动。



虽然大多数数据挖掘工程师对标准化的理解是一致的，但也有一些人要将这种做法区分为“正规化”和“标准化”两种。其中，“正规化”表示将值的范围缩小到0和1之间；“标准化”则是将特征值转换为均值为0的一组数，其中每个数表示偏离均值的程度（即标准偏差或绝对偏差）。我们使用的修正的标准分就是属于后者。

回忆一下，我们上文中有讲解过如何将特征值缩小到0到1之间：找出最大最小值，并做如下计算：

$$\frac{value - \min}{\max - \min}$$

我们来比较一下使用不同的标准化方法得到的准确度：

	classifier built		
data set	using no normalization	using the formula on previous page	using Modified Standard Score
Athletes	80.00%	60.00%	80.00%
Iris	100.00%	83.33%	93.33%
MPG	32.00%	36.00%	56.00%

看来还是使用修正的标准分结果会好些。



用不同的数据集来测试我们的算法是不是很有趣？这些数据集是从UCI机器学习仓库中获得的。去下载一些新的数据集，调整一下格式，测试我们学过的算法吧！

第五章：进一步探索分类

原文：http://guidetodatamining.com/chapter-5

效果评估算法和kNN

让我们回到上一章中运动项目的例子。



在那个例子中，我们编写了一个分类器程序，通过运动员的身高和体重来判断她参与的运动项目——体操、田径、篮球等。

上图中的Marissa Coleman，身高6尺1寸，重160磅，我们的分类器可以正确的进行预测：

```
>>> cl = Classifier('athletesTrainingSet.txt')
>>> cl.classify([73, 160])
'Basketball'
```

对于身高4尺9寸，90磅重的人：

```
>>> cl.classify([59, 90])
'Gymnastics'
```

当我们构建完一个分类器后，应该问以下问题：

- 分类器的准确度如何？
- 结果理想吗？
- 如何与其它分类器做比较？



训练集和测试集

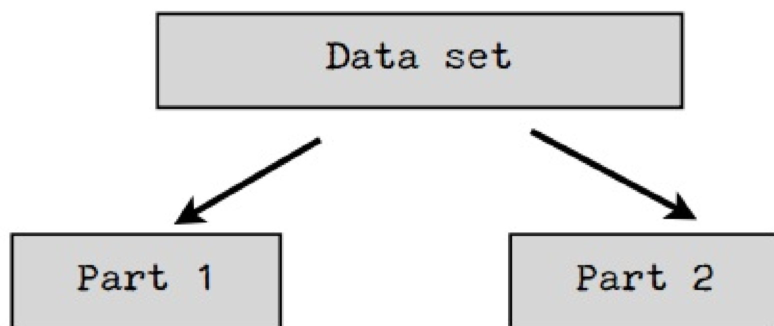
上一章我们一共引入了三个数据集，分别是女运动员、鸢尾花、加仑公里数。我们将这些数据集分为了两个部分，第一部分用来构造分类器，因此称为训练集；另一部分用来评估分类器的结果，因此称为测试集。训练集和测试集在数据挖掘中很常用。

数据挖掘工程师不会用同一个数据集去训练和测试程序。

因为如果使用训练集去测试分类器，得到的结果肯定是百分之百准确的。换种说法，在评价一个数据挖掘算法的效果时，如果用来测试的数据集是训练集本身的一个子集，那结果会极大程度趋向于好，所以这种做法不可取。

将数据集拆分成一大一小两个部分的做法就产生了，前者用来训练，后者用来测试。不过，这种做法似乎也有问题：如果分割的时候不凑巧，就会引发异常。比如，若测试集中的篮球运动员恰巧都很矮，她们就会被归为马拉松运动员；如果又矮又轻，则会被归为体操运动员。使用这样的测试集会造成评分结果非常低。相反的情况也有可能出现，使评分结果趋于100%准确。无论哪种情况发生，都不是一种真实的评价。

解决方法之一是将数据集按不同的方式拆分，测试多次，取结果的平均值。比如，我们将数据集拆为均等的两份：



我们可以先用第一部分做训练集，第二部分做测试集，然后再反过来，取两次测试的平均结果。我们还可以将数据集分成三份，用两个部分来做训练集，一个部分来做测试集，迭代三次：

1. 使用Part 1和Part 2训练，使用Part 3测试；
2. 使用Part 1和Part 3训练，使用Part 2测试；
3. 使用Part 2和Part 3训练，使用Part 1测试；

最后取三次测试的平均结果。

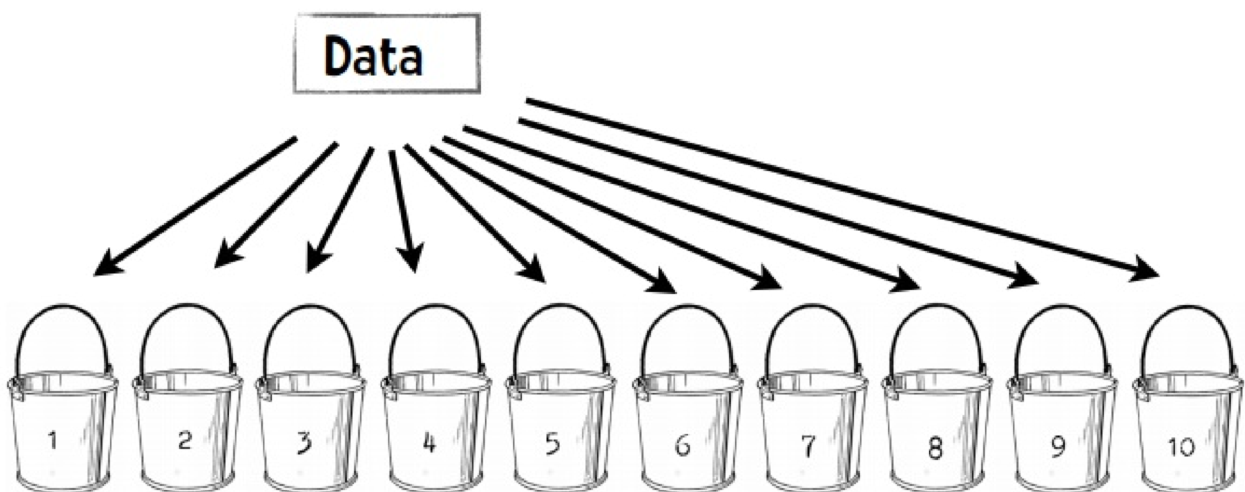
在数据挖掘中，通常的做法是将数据集拆分成十份，并按上述方式进行迭代测试。因此这种方式也称为——

十折交叉验证

将数据集随机分割成十个等份，每次用9份数据做训练集，1份数据做测试集，如此迭代10次。

我们来看一个示例：假设我有一个分类器能判断某个人是否是篮球运动员。我的数据集包含500个运动员和500个普通人。

第一步：将数据分成**10**份



每个桶中会放50个篮球运动员，50个普通人，一共100人。

第二步：重复以下步骤**10**次

1. 每次迭代我们保留一个桶，比如第一次迭代保留木桶1，第二次保留木桶2。
2. 我们使用剩余的9个桶来训练分类器，比如第一次迭代使用木桶2至10来训练。
3. 我们用刚才保留的一个桶来进行测试，并记录结果，比如：35个篮球运动员分类正确，29个普通人分类正确。

第三步：合并结果

我们可以用一张表格来展示结果：

	判定为篮球运动员的人数	判定为普通人的人数
的确是篮球运动员的人数	372	128
的确是普通人的人数	220	280

500个篮球运动员中有372个人判断正确，500个普通人中有280个人判断正确，所以我们可以认为1000人中有652个人判断正确，准确率就是65.2%。通过十折交叉验证得到的评价结果肯定会比二折或者三折来得准确，毕竟我们使用了90%的数据进行训练，而非二折验证中的50%。

> 好，既然十折交叉验证效果那么好，我们为何不做一个N折交叉验证？N即数据集中的数据量。
>
> 如果我们有1000个数据，我们就用999个数据来训练分类器，再用它去判定剩下的一个数据。这样得到的验证效果应该是最好的。



留一法

在数据挖掘领域，N折交叉验证又称为留一法。上面已经提到了留一法的优点之一：我们用几乎所有的数据进行训练，然后用一个数据进行测试。留一法的另一个优点是：确定性。

什么是确定性？

试想Lucy花了一整周的时间编写了一个分类器。周五的时候她请两位同事（Emily和Li）来对这个分类器进行测试，并给了他们相同的数据集。这两位同事都使用十折交叉验证，结果是：



> Emily：这个分类器的准确率是73.69%，很不错！
>
> Li：它的准确率只有71.27%。

为什么她们的结果不一样？是某个人计算发生错误了吗？其实不是。在十折交叉验证中，我们需要将数据随机等分成十份，因此Emily和Li的分法很有可能是不一样的。这样一来，她们的训练集和测试集也都不相同了，得到的结果自然不同。即使是同一个人进行检验，如果两次使用了不同的分法，得到的结果也会有差异。因此，十折交叉验证是一种不确定的验证。相反，留一法得到的结果总是相同的，这是它的一个优点。

留一法的缺点

最大的缺点是计算时间很长。假设我们有一个包含1000条记录的数据集，使用十折交叉验证需要运行10分钟，而使用留一法则需要16个小时。如果我们的数据集更大，达到百万级，那检验的时间就更长了。



> 我两年后再给你检验结果！

留一法的另一个缺点是分层问题。

分层问题

让我们回到运动员分类的例子——判断女运动员参与的项目是篮球、体操、还是田径。在训练分类器的时候，我们会试图让训练集包含全部三种类别。如果我们完全随机分配，训练集中有可能会不包含篮球运动员，在测试的时候就会影响结果。比如说，我们来构建一个包含100个运动员的数据集：从女子NBA网站上获取33名篮球运动员的信息，到Wikipedia上获取33个参加过2012奥运会体操项目的运动员，以及34名田径运动员的信息。这个数据集看起来是这样的：

comment	class	num	num	
Asuka Teramoto		Gymnastics	54	66
Brittainey Raven		Basketball	72	162
Chen Nan	Basketball	78	204	
Gabby Douglas	Gymnastics	49	90	
Helalia Johannes		Track	65	99
Irina Miketenko	Track	63	106	
Jennifer Lacy	Basketball	75	175	
Kara Goucher	Track	67	123	
Linlin Deng	Gymnastics	54	68	
Nakia Sanford	Basketball	76	200	
Nikki Blue	Basketball	68	163	
Qiushuang Huang		Gymnastics	61	95
Rebecca Tunney		Gymnastics	58	77
Rene Kalmer	Track	70	108	
Shanna Crossley		Basketball	70	155
Shavonte Zellous		Basketball	70	155
Tatyana Petrova	Track	63	108	
Tiki Gelana	Track	65	106	
Valeria Straneo	Track	66	97	
Viktoria Komova		Gymnastics	61	76

33名篮球运动员

comment	class	num	num	
Asuka Teramoto		Gymnastics	54	66
Brittainey Raven		Basketball	72	162
Chen Nan	Basketball	78	204	
Gabby Douglas	Gymnastics	49	90	
Helalia Johannes		Track	65	99
Irina Miketenko	Track	63	106	
Jennifer Lacy	Basketball	75	175	
Kara Goucher	Track	67	123	
Linlin Deng	Gymnastics	54	68	
Nakia Sanford	Basketball	76	200	
Nikki Blue	Basketball	68	163	
Qiushuang Huang		Gymnastics	61	95
Rebecca Tunney		Gymnastics	58	77
Rene Kalmer	Track	70	108	
Shanna Crossley		Basketball	70	155
Shavonte Zellous		Basketball	70	155
Tatyana Petrova	Track	63	108	
Tiki Gelana	Track	65	106	
Valeria Straneo	Track	66	97	
Viktoria Komova		Gymnastics	61	76

33名体操运动员

comment	class	num	num	
Asuka Teramoto		Gymnastics	54	66
Brittainey Raven		Basketball	72	162
Chen Nan	Basketball	78	204	
Gabby Douglas	Gymnastics	49	90	
Helalia Johannes		Track	65	99
Irina Miketenko	Track	63	106	
Jennifer Lacy	Basketball	75	175	
Kara Goucher	Track	67	123	
Linlin Deng	Gymnastics	54	68	
Nakia Sanford	Basketball	76	200	
Nikki Blue	Basketball	68	163	
Qiushuang Huang		Gymnastics	61	95
Rebecca Tunney		Gymnastics	58	77
Rene Kalmer	Track	70	108	
Shanna Crossley		Basketball	70	155
Shavonte Zellous		Basketball	70	155
Tatyana Petrova	Track	63	108	
Tiki Gelana	Track	65	106	
Valeria Straneo	Track	66	97	
Viktoria Komova		Gymnastics	61	76

34名马拉松运动员

现在我们来做十折交叉验证。我们按顺序将这些运动员放到10个桶中，所以前三个桶放的都是篮球运动员，第四个桶有篮球运动员也有体操运动员，以此类推。这样一来，没有一个桶能真正代表这个数据集的全貌。最好的方法是将不同类别的运动员按比例分发到各个桶中，这样每个桶都会包含三分之一篮球运动员、三分之一体操运动员、以及三分之一田径运动员。这种做法叫做分层。而在留一法中，所有的测试集都只包含一个数据。所以说，留一法对小数据集是合适的，但大多数情况下我们会选择十折交叉验证。

混淆矩阵



目前我们衡量分类器准确率的方式是使用以下公式：正确分类的记录数÷记录总数。有时我们会需要一个更为详细的评价结果，这时就会用到一个称为混淆矩阵的可视化表格。表格的行表示测试用例实际所属的类别，列则表示分类器的判断结果。

混淆矩阵可以帮助我们快速识别出分类器到底在哪些类别上发生了混淆，因此得名。让我们看看运动员的示例，这个数据集中有300人，使用十折交叉验证，其混淆矩阵如下：

	体操	篮球	马拉松
体操	83	0	17
篮球	0	92	8
马拉松	9	16	85

可以看到，100个体操运动员中有83人分类正确，17人被错误地分到了马拉松一列；92个篮球运动员分类正确，8人被分到了马拉松；85个马拉松运动员分类正确，9人被分到了体操，16人被分到了篮球。

混淆矩阵的对角线（绿色字体）表示分类正确的人数，因此求得的准确率是：

$$\frac{83+92+85}{300}=\frac{260}{300}=86.66\%$$

从混淆矩阵中可以看出分类器的主要问题。在这个示例中，我们的分类器可以很好地区分体操运动员和篮球运动员，而马拉松运动员则比较容易和其他两个类别发生混淆。



> 怎样，是不是觉得混淆矩阵其实并不混淆呢？

代码示例

让我们使用加仑公里数这个数据集，格式如下：

mpg	cylinders	c.i.	HP	weight	secs. 0-60	make/model
30	4	68	49	1867	19.5	fiat 128
45	4	90	48	2085	21.7	vw rabbit (diesel)
20	8	307	130	3504	12	chevrolet chevelle malibu

我会通过汽车的以下属性来判断它的加仑公里数：汽缸数、排气量、马力、重量、加速度。我将392条数据都存放在mpgData.txt文件中，并用下面这段Python代码将这些数据按层次等分成十份：

```
# -*- coding: utf-8 -*-

# 将数据等分成十份的示例代码

import random

def buckets(filename, bucketName, separator, classColumn):
    """filename是源文件名
    bucketName是十个目标文件的前缀名
    separator是分隔符，如制表符、逗号等
    classColumn是表示数据所属分类的那一列的序号"""

    # 将数据分为10份
    numberOfBuckets = 10
    data = {}
    # 读取数据，并按分类放置
    with open(filename) as f:
        lines = f.readlines()
    for line in lines:
        if separator != '\t':
            line = line.replace(separator, '\t')
        # 获取分类
        category = line.split()[classColumn]
        data.setdefault(category, [])
        data[category].append(line)
    # 初始化分桶
    buckets = []
    for i in range(numberOfBuckets):
        buckets.append([])
    # 将各个类别的数据均匀地放置到桶中
    for k in data.keys():
        # 打乱分类顺序
        random.shuffle(data[k])
        bNum = 0
        # 分桶
        for item in data[k]:
            buckets[bNum].append(item)
            bNum = (bNum + 1) % numberOfBuckets

    # 写入文件
    for bNum in range(numberOfBuckets):
        f = open("%s-%02i" % (bucketName, bNum + 1), 'w')
        for item in buckets[bNum]:
            f.write(item)
        f.close()

# 调用示例
buckets("pimaSmall.txt", 'pimaSmall',',',8)
```

执行这个程序后会生成10个文件：mpgData01、mpgData02等。

编程实践

你能否修改上一章的近邻算法程序，让 `test` 函数能够执行十折交叉验证？输出的结果应该是这样的：

		predicted MPG							
		10	15	20	25	30	35	40	45
actual MPG	10	3	10	0	0	0	0	0	0
	15	3	68	14	1	0	0	0	0
	20	0	14	66	9	5	1	1	0
	25	0	1	14	35	21	6	1	1
	30	0	1	3	17	21	14	5	2
	35	0	0	2	8	9	14	4	1
	40	0	0	1	0	5	5	0	0
	45	0	0	0	2	1	1	0	2

53.316% accurate
total of 392 instances

解决方案

我们需要进行以下几步：

- 修改初始化方法，只读取九个桶中的数据作为训练集；
- 增加一个方法，从第十个桶中读取测试集；
- 执行十折交叉验证。

下面我们分步来看：

- 初始化方法 `__init__`

`__init__` 方法的签名会修改成以下形式：

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat):
```

每个桶的文件名是mpgData-01、mpgData-02这样的形式，所以 `bucketPrefix` 就是“mpgData”。 `testBucketNumber` 是测试集所用的桶，如果是3，则分类器会使用1、2、4-9的桶进行训练。 `dataFormat` 用来指定数据集的格式，如：

```
class    num    num    num    num    num    comment
```

意味着第一列是所属分类，后五列是特征值，最后一列是备注信息。

以下是初始化方法的示例代码：

```
class Classifier:

    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):
        """该分类器程序将从bucketPrefix指定的一系列文件中读取数据，
        并留出testBucketNumber指定的桶来做测试集，其余的做训练集。
        dataFormat用来表示数据的格式，如：
        "class    num    num    num    num    num    comment"
    """

        self.medianAndDeviation = []

        # 从文件中读取文件

        self.format = dataFormat.strip().split('\t')
        self.data = []
        # 用1-10来标记桶
        for i in range(1, 11):
            # 判断该桶是否包含在训练集中
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                f = open(filename)
                lines = f.readlines()
                f.close()
                for line in lines[1:]:
                    fields = line.strip().split('\t')
                    ignore = []
                    vector = []
                    for i in range(len(fields)):
                        if self.format[i] == 'num':
                            vector.append(float(fields[i]))
                        elif self.format[i] == 'comment':
                            ignore.append(fields[i])
                        elif self.format[i] == 'class':
                            classification = fields[i]
                    self.data.append((classification, vector, ignore))
        self.rawData = list(self.data)
        # 获取特征向量的长度
        self.vlen = len(self.data[0][1])
        # 标准化数据
        for i in range(self.vlen):
            self.normalizeColumn(i)
```

- testBucket方法

下面的方法会使用一个桶的数据进行测试：

```
def testBucket(self, bucketPrefix, bucketNumber):
    """读取bucketPrefix-bucketNumber所指定的文件作为测试集"""

    filename = "%s-%02i" % (bucketPrefix, bucketNumber)
    f = open(filename)
    lines = f.readlines()
    totals = {}
    f.close()
    for line in lines:
        data = line.strip().split('\t')
        vector = []
        classInColumn = -1
        for i in range(len(self.format)):
              if self.format[i] == 'num':
                  vector.append(float(data[i]))
              elif self.format[i] == 'class':
                  classInColumn = i
        theRealClass = data[classInColumn]
        classifiedAs = self.classify(vector)
        totals.setdefault(theRealClass, {})
        totals[theRealClass].setdefault(classifiedAs, 0)
        totals[theRealClass][classifiedAs] += 1
    return totals
```

比如说bucketPrefix是mpgData，bucketNumber是3，那么程序会从mpgData-03中读取内容，作为测试集。这个方法会返回如下形式的结果：

```
{'35': {'35': 1, '20': 1, '30': 1},
 '40': {'30': 1},
 '30': {'35': 3, '30': 1, '45': 1, '25': 1},
 '15': {'20': 3, '15': 4, '10': 1},
 '10': {'15': 1},
 '20': {'15': 2, '20': 4, '30': 2, '25': 1},
 '25': {'30': 5, '25': 3}}
```

这个字段的键表示真实类别。如第一行的35表示该行数据的真实类别是35加仑公里。这个键又对应一个字典，这个字典表示的是分类器所判断的类别，如：

```
'15': {'20': 3, '15': 4, '10': 1},
```

其中的3表示有3条记录真实类别是15加仑公里，但被分类到了20加仑公里；4表示分类正确的记录数；1表示被分到10加仑公里的记录数。

- 执行十折交叉验证

最后我们需要编写一段程序来执行十折交叉验证，也就是说要用不同的训练集和测试集来构建10个分类器。

```
def tenfold(bucketPrefix, dataFormat):
    results = {}
    for i in range(1, 11):
        c = Classifier(bucketPrefix, i, dataFormat)
        t = c.testBucket(bucketPrefix, i)
        for (key, value) in t.items():
            results.setdefault(key, {})
            for (ckey, cvalue) in value.items():
                results[key].setdefault(ckey, 0)
                results[key][ckey] += cvalue

    # 输出结果
    categories = list(results.keys())
    categories.sort()
    print(   "\n       Classified as: ")
    header =    "         "
    subheader = "       +"
    for category in categories:
        header += category + "    "
        subheader += "----+"
    print (header)
    print (subheader)
    total = 0.0
    correct = 0.0
    for category in categories:
        row = category + "     |"
        for c2 in categories:
            if c2 in results[category]:
                count = results[category][c2]
            else:
                count = 0
            row += " %2i |" % count
            total += count
            if c2 == category:
                correct += count
        print(row)
    print(subheader)
    print("\n%5.3f percent correct" %((correct * 100) / total))
    print("total of %i instances" % total)

# 调用方法
tenfold("mpgData/mpgData",        "class     num     num     num     num     num     comment")
```

执行结果如下：

```
         Classified as:
          10   15   20   25   30   35   40   45
       +----+----+----+----+----+----+----+----+
10     |  3 | 10 |  0 |  0 |  0 |  0 |  0 |  0 |
15     |  3 | 68 | 14 |  1 |  0 |  0 |  0 |  0 |
20     |  0 | 14 | 66 |  9 |  5 |  1 |  1 |  0 |
25     |  0 |  1 | 14 | 35 | 21 |  6 |  1 |  1 |
30     |  0 |  1 |  3 | 17 | 21 | 14 |  5 |  2 |
35     |  0 |  0 |  2 |  8 |  9 | 14 |  4 |  1 |
40     |  0 |  0 |  1 |  0 |  5 |  5 |  0 |  0 |
45     |  0 |  0 |  0 |  2 |  1 |  1 |  0 |  2 |
       +----+----+----+----+----+----+----+----+

53.316 percent correct
total of 392 instances
```

可以在这里下载代码和数据集。

Kappa指标

本章的开头我们对分类器的效果提了几个问题，并在此之后使用十折交叉验证和混淆矩阵来对分类器进行评估。上一节中我们对加仑公里数分类器的评价结果是53.316%的正确率，那这个结果是好是坏呢？我们就需要使用一个新的指标：Kappa指标。



Kappa指标可以用来评价分类器的效果比随机分类要好多少。我们仍用运动员的例子来说明，以下是它的混淆矩阵：

	体操	篮球	马拉松	合计
体操	35	5	20	60
篮球	0	88	12	100
马拉松	5	7	28	40
合计	40	100	60	200

我增加了“合计”一列，因此在计算正确率时，我们只需将对角线相加（35 + 88 + 28 = 151）除以合计（200）就可以了，结果是0.755。

现在，我们建造另一个混淆矩阵，用来表示随机分类的结果。首先，我们将上表中的数据抹去一部分，只留下合计：

	体操	篮球	马拉松	合计
体操				60
篮球				100
马拉松				40
合计	40	100	60	200

从最后一行可以看到，我们之前构造的分类器将50%的运动员分类到篮球运动员中（200中的100人），20%分到了体操，剩余30%分到了马拉松。即：

- 体操 20%
- 篮球 50%
- 田径 30%

我们会用这个百分比来构造随机分类器的混淆矩阵。比如，真实的体操运动员一共有60人，随机分类器会将其中的20%（12人）分类为体操，50%（30人）分类为篮球，30%（18人）分类为马拉松，填入表格：

	体操	篮球	马拉松	合计
体操	12	30	18	60
篮球				100
马拉松				40
合计	40	100	60	200

继续用这种方法填充空白。100个真实的篮球运动员，20%（20人）分到体操，50%（50人）分到篮球，30%（30人）分到马拉松。

	体操	篮球	马拉松	合计
体操	12	30	18	60
篮球	20	50	30	100
马拉松	8	20	12	40
合计	40	100	60	200

从而得到随机分类器的准确率是：

$$P(r) = \frac{12 + 50 + 12}{200} = \frac{74}{200} = .37$$

Kappa指标可以用来衡量我们之前构造的分类器和随机分类器的差异，公式为：

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)}$$

P(c)表示分类器的准确率，P(r)表示随机分类器的准确率。将之前的结果代入公式：

$$\kappa = \frac{.755 - .37}{1 - .37} = \frac{.385}{.63} = .61$$

0.61要如何解释呢？可以参考下列经验结果：

<0	比随机分类要差
0.01－0.20	稍好
0.21－0.40	一般
0.41－0.60	符合期望
0.61－0.80	本质上超越
0.81－1.00	几近完美

来源：*Landis, JR, Koch, GG. 1977* 分类效果评估 生物测量学

动手实践

假设我们开发了一个效果不太好的分类器，用来判断600名大学生所读专业，使用的数据是他们对10部电影的评价。这些大学生的专业类别有计算机科学（cs）、教育学（ed）、英语（eng）、心理学（psych）。以下是该分类器的混淆矩阵，尝试计算出它的Kappa指标并予以解释。

	cs	ed	eng	psych	Total
cs	50	8	15	7	
ed	0	75	12	33	
eng	5	12	123	30	
psych	5	25	30	170	

准确率 = 0.697

解答

首先，计算列合计和百分比：

	cs	ed	eng	psych	TOTAL
SUM	60	120	180	240	600
%	10%	20%	30%	40%	100%

然后根据百分比来填充随机分类器的混淆矩阵：

	cs	ed	eng	psych	Total
cs	8	16	24	32	80
ed	12	24	36	48	120
eng	17	34	51	68	170
psych	23	46	69	92	230
Total	60	120	180	240	600

准确率 = (8 + 24 + 51 + 92) / 600 = (175 / 600) = 0.292

最后，计算Kappa指标：

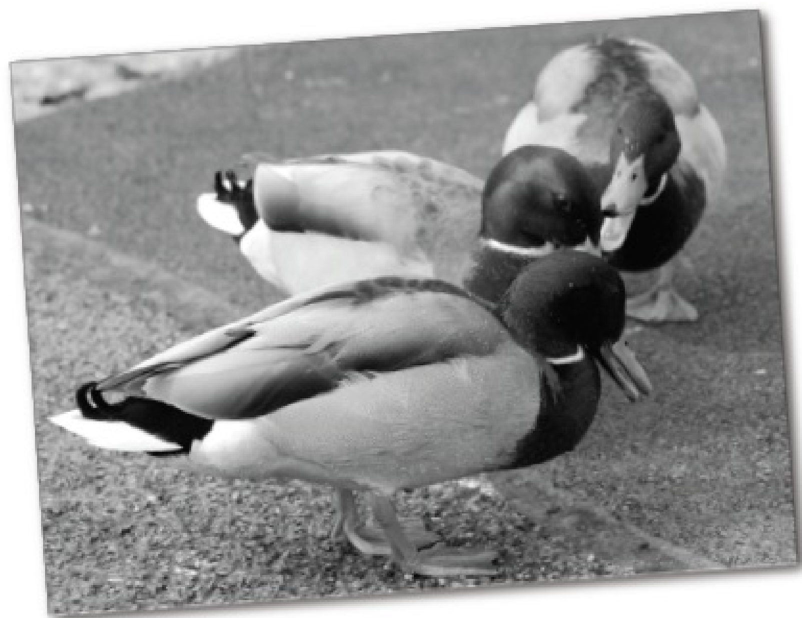
$$\kappa = \frac{0.697 - 0.292}{1 - 0.292} = \frac{0.405}{0.708} = 0.572$$

这说明分类器的效果还是要好过预期的。

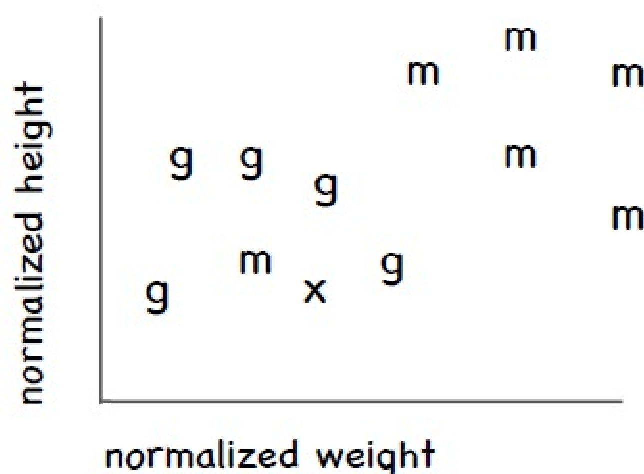


优化近邻算法

有一种分类器叫“机械记忆分类器（Rote Classifer）”，它会将数据集完整地保存下来，并用来判断某条记录是否存在于数据集中。所以，如果我们只对数据集中的数据进行分类，准确率将是100%。而在现实应用中，这种分类器并不可用，因为我们需要判定某条新的记录属于哪个分类。你可以认为我们上一章中构建的分类器是机械记忆分类器的一种扩展，只是我们不要求新的记录完全对应到数据集中的某一条记录，只要距离最近就可以了。PangNing Tan等人在其机器学习的教科书中写过这样一段话：如果一只动物走起来像鸭子、叫起来像鸭子、而且看起来也像鸭子，那它很有可能就是一只鸭子。



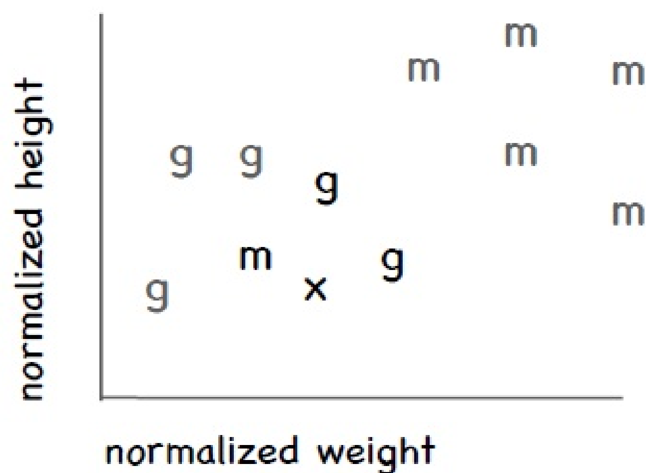
近邻算法的问题之一是异常数据。还是拿运动员分类举例，不过只看体操和马拉松。假设有一个比较矮也比较轻的人，她是马拉松运动员，这样就会形成以下这张图，横轴是体重，纵轴是身高，其中m表示马拉松，g表示体操：



可以看到这名特别的马拉松运动员处于体操运动员的群组中。假设我们要对一名新的运动员进行分类，用图中的x表示，可以看到离她最近的运动员是那名特别的马拉松运动员，这样一来这名新的运动员就会被归到马拉松，但实际上她更有可能是一名体操运动员。

kNN算法

优化方法之一是考察这条新记录周围距离最近的k条记录，而不是只看一条，因此这种方法称为k近邻算法（kNN）。每个近邻都有投票权，程序会将新记录判定为得票数最多的分类。比如说，我们使用三个近邻（k = 3），其中两条记录属于体操，一条记录属于马拉松，那我们会判定x为体操。





因此，我们在判定一条记录的具体分类时可以用这种投票的方法。如果两个分类的票数相等，就随机选取一个。但对于需要预测具体数值的情形，比如一个人对Funky Meters乐队的评分，我们可以计算k个近邻的距离加权平均值。举个例子，我们需要预测Ben对Funky

Meters的喜好程度，他的三个近邻分别是Sally、Tara、和Jade。下表是这三个人离Ben的距离，以及他们对Funky Meters的评分：

用户	距离	评分
Sally	5	4
Tara	10	5
Jade	15	5

可以看到，Sally离Ben最近，她给Funky Meters的评分是4。在计算平均值的时候，我希望距离越近的用户影响越大，因此可以对距离取倒数，从而得到下表：

用户	距离的倒数	评分
Sally	0.2	4
Tara	0.1	5
Jade	0.067	5

下面，我们把所有的距离倒数除以距离倒数的和（0.2 + 0.1 + 0.067 = 0.367），从而得到评分的权重：

用户	权重	评分
Sally	0.545	4
Tara	0.272	5
Jade	0.183	5

我们可以注意到两件事情：权重之和是1；原始数据中，Sally的距离是Tara的二分之一，这点在权重中体现出来了。最后，我们求得平均值，也即预测Ben对Funky Meters的评分：

$$= 0.545 \times 4 + 0.272 \times 5 + 0.183 \times 5$$

$$= 2.18 + 1.36 + 0.915 = 4.455$$

动手实践

我们想要预测Sofia对爵士钢琴手Hiromi的评分，以下是她三个近邻的距离和评分：

person	distance From Sofia	rating For Hiromi
Gabriela	4	3
Ethan	8	3
Jayden	10	5

解答

第一步，计算距离的倒数：

Person	Inverse Distance	Rating
Gabriela	1/4 = 0.25	3
Ethan	1/8 = 0.125	3
Jayden	1/10 = 0.1	5

第二步，计算权重（距离倒数之和为0.475）：

Person	Influence	Rating
Gabriela	0.526	3
Ethan	0.263	3
Jayden	0.211	5

第三步，预测评分：

$$=(0.526\times 3)+(0.263\times 3)+(0.211\times 5)$$

$$=1.578+0.789+1.055=3.422$$

新的数据集，新的挑战！

是时候使用新的数据集了——比马印第安人糖尿病数据集，由美国国家糖尿病、消化和肾脏疾病研究所提供。



令人惊讶的是，超过30%的比马人患有糖尿病，而全美的糖尿病患者比例是8.3%，中国只有4.2%。

数据集中的一条记录代表一名21岁以上的比马女性，她们分类两类：五年内查出患有糖尿病，以及没有得病。共选取了8个特征：

1. 怀孕次数；
2. 口服葡萄糖耐量实验两小时后的血浆葡萄糖浓度；
3. 舒张压（mm Hg）；
4. 三头肌皮褶厚度（mm）；
5. 血清胰岛素（mu U/ml）；
6. 身体质量指数（BMI）：体重（公斤）除以身高（米）的平方；
7. 糖尿病家谱；
8. 年龄（岁）。

以下是示例数据，最后一列的0表示没有糖尿病，1表示患有糖尿病：

2	99	52	15	94	24.6	0.637	21	0
3	83	58	31	18	34.3	0.336	25	0
5	139	80	35	160	31.6	0.361	25	1
3	170	64	37	225	34.5	0.356	30	1

比如说，第一条记录表示一名生过两次小孩的女性，她的血糖浓度是99，舒张压是52，等等。



实践[1]

本书提供了两份数据集：pimaSmall.zip和pima.zip。前者包含100条记录，后者包含393条记录，都已经等分成了10个文件（10个桶）。我用前面实现的近邻算法计算了pimaSmall数据集，得到的结果如下：

```
         Classified as:
          0    1
        +----+----+
0       | 45 | 14 |
1       | 27 | 14 |
        +----+----+

59.000 percent correct
total of 100 instances
```

提示：代码中的*heapq.nsmallest(n, list)*会返回*n*个最小的项。*heapq*是*Python*内置的优先队列类库。

你的任务是实现kNN算法。首先在类的init函数中添加参数k：

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat, k):
```

knn函数的签名应该是：

```
def knn(self, itemVector):
```

它会使用到 `self.k` （记得在init函数中保存这个值），它的返回值是0或1。此外，在进行十折交叉验证（tenFold函数）时也要传入k参数。

解答

init函数的修改很简单：

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat, k):
    self.k = k
    ...
```

knn函数的实现是：

```
def knn(self, itemVector):
    """使用kNN算法判断itemVector所属类别"""
    # 使用heapq.nsmallest来获得k个近邻
    neighbors = heapq.nsmallest(self.k,
                                [(self.manhattan(itemVector, item[1]), item)
                                 for item in self.data])
    # 每个近邻都有投票权
    results = {}
    for neighbor in neighbors:
        theClass = neighbor[1][0]
        results.setdefault(theClass, 0)
        results[theClass] += 1
    resultList = sorted([(i[1], i[0]) for i in results.items()], reverse=True)
    # 获取得票最高的分类
    maxVotes = resultList[0][0]
    possibleAnswers = [i[1] for i in resultList if i[0] == maxVotes]
    # 若得票相等则随机选取一个
    answer = random.choice(possibleAnswers)
    return(answer)
```

对tenFold函数的改动如下：

```
def tenfold(bucketPrefix, dataFormat, k):
    results = {}
    for i in range(1, 11):
        c = Classifier(bucketPrefix, i, dataFormat, k)
        ...
```

你可以从网站上下载这些代码，不过我的代码并不一定是最优的，仅供参考。

实践**[2]**

在分类效果上，究竟是数据量的多少比较重要（即使用pimaSmall和pima数据集的效果），还是更好的算法比较重要（k=1和k=3）？

解答

以下是比较结果：

	pimaSmall	pima
k=1	59.00%	71.247%
k=3	61.00%	72.519%

看来增加数据量要比使用更好的算法带来的效果好。



实践**[3]**

72.519%的准确率看起来不错，但还是用Kappa指数来检验一下吧：

	健康	患有糖尿病
健康	219	44
患有糖尿病	64	66

解答

计算合计和比例：

	健康	患有糖尿病	合计
健康	219	44	263
患有糖尿病	64	66	130
合计	283	110	393
比例	0.7201	0.2799	

随机分类器的混淆矩阵：

	健康	患有糖尿病
健康	189.39	73.61
患有糖尿病	93.61	36.39

随机分类器的正确率：

$$p(r) = \frac{189.39 + 36.61}{393}$$
$$= .5745$$

Kappa指标：

$$\kappa = \frac{P(c)-P(r)}{1-P(r)} = \frac{.72519-.5745}{1-.5745} = \frac{.15069}{.4255} = .35415$$

效果一般

更多数据，更好的算法，还有抛锚的巴士



几年前我去参加一个墨西哥城的研讨会，比较特别的是会议的第二天是坐观光巴士旅游，观看黑脉金斑蝶等。这辆巴士比较破旧，中途抛锚了好几次，所以我们一群有着博士学位的人就站在路边一边谈笑，一边等着司机修理巴士。而事实证明这段经历是这次会议最大的收获。其间，我有幸与Eric Brill做了交流，他在词性分类方面有着很高的成就，他的算法比前人要优秀很多，从而使他成为自然语言处理界的名人。我和他谈论了分类器的效果问题，他说实验证明增加数据所带来的效果要比改进算法来得大。事实上，如果仍沿用老的词性分类算法，而仅仅增加训练集的数据量，效果很有可能比他现有的算法更好。当然，他不可能通过收集更多的数据来获得一个博士学位，但如果如果你的算法能够取得哪怕一点点改进，也足够了。

更多数据 ⇨ Més dades ⇨ More data

当然，这并不是说你就不需要挑选出更好的算法了，我们之前也看到了好的算法所带来的效果也是惊人的。但是如果你只是想解决一个问题，而非发表一篇论文，那增加数据量会更经济一些。

所以，在认同数据量多寡的重要影响后，我们仍将继续学习各种算法。

人们使用**kNN**算法来做以下事情：

- 在亚马逊上推荐商品
- 评估用户的信用
- 通过图像分析来分类路虎车型
- 人像识别

- 分析照片中人物的性别
- 推荐网页
- 推荐旅游套餐

第六章：概率和朴素贝叶斯

原文：http://guidetodatamining.com/chapter-6

朴素贝叶斯

还是让我们回到运动员的例子。如果我问你Brittney Griner的运动项目是什么，她有6尺8寸高，207磅重，你会说“篮球”；我再问你对此分类的准确度有多少信心，你会回答“非常有信心”。

我再问你Heather Zurich，6尺1寸高，重176磅，你可能就不能确定地说她是打篮球的了，至少不会像之前判定Brittney那样肯定。因为从Heather的身高体重来看她也有可能是跑马拉松的。

最后，我再问你Yumiko Hara的运动项目，她5尺4寸高，95磅重，你也许会说她是跳体操的，但也不太敢肯定，因为有些马拉松运动员也是类似的身高体重。



使用近邻算法时，我们很难对分类结果的置信度进行量化。但如果使用的是基于概率的分类算法——贝叶斯算法——那就可以给出分类结果的可能性了：这名运动员有80%的几率是篮球运动员；这位病人有40%的几率患有糖尿病；拉斯克鲁塞斯24小时内有雨的概率是10%。

近邻算法又称为被动学习算法。这种算法只是将训练集的数据保存起来，在收到测试数据时才会进行计算。如果我们有10万首音乐，那每进行一次分类，都需要遍历这10万条记录才行。



贝叶斯算法则是一种主动学习算法。它会根据训练集构建起一个模型，并用这个模型来对新的记录进行分类，因此速度会快很多。



所以说，贝叶斯算法的两个优点即：能够给出分类结果的置信度；以及它是一种主动学习算法。

概率

相信大多数人对概率并不陌生。比如投掷一个硬币，正面出现的概率是50%；掷骰子，出现1点的概率是16.7%；从一群19岁的青少年中随机挑出一个，让你说出她是女生的可能性，你会回答50%。以上这些我们用符号P(h)来表示，即事件h发生的概率：

- 投掷硬币：P(正面) = 0.5
- 掷骰子：P(1) = 1/6
- 青少年：P(女生) = 0.5

如果我再告诉你一些额外的信息，比如这群19岁的青少年都是弗兰科学院建筑专业的学生，于是你到Google上搜索后发现这所大学的女生占86%，这时你就会改变你的答案——女生的可能性是86%。

这一情形我们用P(h|D)来表示，即D条件下事件h发生的概率。比如：

P(女生|弗兰克学院的学生) = 0.86

计算的公式是：

$$P(A|B)=\frac{P(A\cap B)}{P(B)}$$

再举一个例子，下表是一些人使用笔记本电脑和手机的品牌：

name	laptop	phone
Kate	PC	Android
Tom	PC	Android
Harry	PC	Android
Annika	Mac	iPhone
Naomi	Mac	Android
Joe	Mac	iPhone
Chakotay	Mac	iPhone
Neelix	Mac	Android
Kes	PC	iPhone
B'Elanna	Mac	iPhone

使用iPhone的概率是多少？

$$P(iPhone)=\frac{5}{10}=0.5$$

如果已知这个人使用的是Mac笔记本，那他使用iPhone的概率是？

$$P(iPhone|mac)=\frac{P(mac\cap iPhone)}{P(mac)}$$

首先计算出同时使用Mac和iPhone的概率：

$$P(mac\cap iPhone)=\frac{4}{10}=0.4$$

使用Mac的概率则是：

$$P(mac) = \frac{6}{10} = 0.6$$

从而计算得到Mac用户中使用iPhone的概率：

$$P(iPhone \mid mac) = \frac{0.4}{0.6} = 0.667$$

以上是正规的解法，不过为了简单起见，我们可以直接通过计数得到：

$$\text{P(iPhone|mac)} = \frac{\text{同时使用 Mac 和 iPhone 的人数}}{\text{使用 Mac 的人数}}$$

$$\text{P(iPhone|mac)} = \frac{4}{6} = 0.667$$

练习

iPhone用户中使用Mac的概率是？

$$P(mac \mid iPhone) = \frac{P(iPhone \cap mac)}{P(iPhone)}$$

$$= \frac{0.4}{0.5} = 0.8$$



术语

P(h)表示事件h发生的概率，称为h的先验概率。在我们进行任何计算之前就已经得知人们使用Mac的概率是0.6。计算之后我们可能会得知使用Mac的人同时会使用iPhone。

P(h|d)称为后验概率，表示在观察了数据集d之后，h事件发生的概率是多少。比如说，我们在观察了使用iPhone的用户后可以得出他们使用Mac的概率是多少。后验概率又称为条件概率。

在构建一个贝叶斯分类器前，我们还需要两个概率：P(D)和P(D|h)，请看下面的示例。

微软购物车

你听说过微软的智能购物车吗？没错，他们真有这样的产品。这个产品是微软和一个名为Chaotic Moon的公司合作开发的。这家公司的标语是“我们比你聪明，我们比你有创造力。”你可以会觉得这样的标语有些狂妄自大，这里暂且不谈。这种购物车由以下几个部分组成：Windows 8平板电脑、Kinect体感设备、蓝牙耳机（购物车可以和你说话）、以及电动装置（购物车可以跟着你走）。

你走进一家超市，持有一张会员卡，智能购物车会识别出你，它会记录你的购物记录（当然也包括其他人的）。



智能购物车也会显示广告（比如日本的Sensha绿茶），不过它只会向那些有可能购买此物品的用户进行展示。

以下是一些数据示例：

客户 ID	邮编	是否购买有机食品	是否购买 Sencha 绿茶
1	88005	是	是
2	88001	否	否
3	88001	是	是
4	88005	否	否
5	88003	是	否
6	88005	否	是
7	88005	否	否
8	88001	否	否
9	88005	是	是
10	88003	是	是

P(D)表示从训练集数据中计算得到的概率，比如上表中邮编为88005的概率是：

P(88005) = 0.5

P(D|h)表示在一定条件下的观察结果。比如说购买过Sencha绿茶的人中邮编为88005的概率为：

$$P(88005 \mid SenchaTea) = \frac{3}{5} = 0.6$$

练习

没有买Sencha的人中邮编为88005的概率是？

$$P(88005 \mid \neg SenchaTea) = \frac{2}{5} = 0.4$$

上式中的“┐”表示取反

邮编为88001的概率是？

P(88001) = 0.3

购买了Sencha的人中邮编为88001的概率？

$$P(88001 \mid SenchaTea) = \frac{1}{5} = 0.2$$

没有购买Sencha的人中邮编为88001的概率？

$$P(88001 \mid \neg SenchaTea) = \frac{2}{5} = 0.4$$

贝叶斯法则

贝叶斯法则描述了P(h)、P(h|D)、P(D)、以及P(D|h)这四个概率之间的关系：

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)}$$

这个公式是贝叶斯方法论的基石。在数据挖掘中，我们通常会使用这个公式去判别不同事件之间的关系。我们可以计算得到在某些条件下这位运动员是从事体操、马拉松、还是篮球项目的；也可以计算得到某些条件下这位客户是否会购买Sencha绿茶等。我们会通过计算不同事件的概率来得出结论。

比如说我们要决定是否给一位客户展示Sencha绿茶的广告，已知他所在的地区邮编是88005。我们有两个相反的假设：

- 这位用户会购买Sencha绿茶的概率，即：P(购买|88005)；
- 不会购买的概率：P(┐购买|88005)。

假设我们计算得到P(购买|88005) = 0.6，而P(┐购买|88005) = 0.4，则可以认为用户会购买，从而显示相应的广告。

再比如我们要为一家销售电子产品的公司发送宣传邮件，共有笔记本、台式机、平板电脑三种产品。我们需要根据目标用户的类型来分别派送这三种宣传邮件。比如我们有一位居住在88005地区的女士，她的女儿在读大学，并居住在家中，而且她还会参加瑜伽课程。那我应该派发哪种邮件呢？

让我们用D来表示这位客户的特征：

- 居住在88005地区
- 有一个正在读大学的女儿
- 练习瑜伽

因此我们需要计算以下三个概率：

$$P(\text{笔记本}|D) = \frac{P(D|\text{笔记本})P(\text{笔记本})}{P(D)}$$

$$P(\text{台式机}|D) = \frac{P(D|\text{台式机})P(\text{台式机})}{P(D)}$$

$$P(\text{平板电脑}|D) = \frac{P(D|\text{平板电脑})P(\text{平板电脑})}{P(D)}$$

并选择概率最大的结果。

再抽象一点，如果我们有h_1, h_2, ...h_n等事件，它们就相当于不同的类别（篮球、体操、或是有没有患糖尿病等）。

$$P(h_1 | D) = \frac{P(D | h_1)P(h_1)}{P(D)} \quad , \quad P(h_2 | D) = \frac{P(D | h_2)P(h_2)}{P(D)}$$

$$\ldots \quad P(h_n | D) = \frac{P(D | h_n)P(h_n)}{P(D)}$$

在计算出以上这些概率后，选取最大的结果，就能用作分类了。这种方法叫最大后验估计，记为h_{MAP}。



我们可以用以下公式来表示最大后验估计：

$$h_{MAP} = \arg\max_{h \in H} P(h \mid D)$$

H表示所有的事件，所以h∈H表示“对于集合中的每一个事件”。整个公式的含义就是：对于集合中的每一个事件，计算出P(h|D)的值，并取最大的结果。使用贝叶斯法则进行替换：

$$h_{MAP} = \arg\max_{h \in H} \frac{P(D \mid h)P(h)}{P(D)}$$

所以我们需要计算的就是以下这个部分：

$$\frac{P(D \mid h)P(h)}{P(D)}$$

可以发现对于所有的事件，公式中的分母都是P(D)，因此即便只计算P(D|h)P(h)，也可以判断出最大的结果。那么这个公式就可以简化为：

$$h_{MAP} = \arg\max_{h \in H} P(D \mid h)P(h)$$

作为演示，我们选取Tom M. Mitchell《机器学习》中的例子。Tom是卡耐基梅隆大学机器学习部的首席，也是非常友好的一个人。这个例子是通过一次血液检验来判断某人是否患有某种癌症。已知这种癌症在美国的感染率是0.8%。血液检验的结果有阳性和阴性两种，且存在准确性的问题：如果这个人患有癌症，则有98%的几率测出阳性；如果他没有癌症，会有97%的几率测出阴性。

我们来尝试将这些描述语言用公式来表示：

- 美国有0.8%的人患有这种癌症：P(癌症) = 0.008
- 99.2%的人没有患有这种癌症：P(┐癌症) = 0.992

- 对于患有癌症的人，他的血液检测结果返回阳性的概率是98%：P(阳性|癌症) = 0.98
- 对于患有癌症的人，检测结果返回阴性的概率是2%：P(阴性|癌症) = 0.02
- 对于没有癌症的人，返回阴性的概率是97%：P(阴性|┐癌症) = 0.97
- 对于没有癌症的人，返回阳性的概率是3%：P(阳性|┐癌症) = 0.03



Ann到医院做了血液检测，呈阳性。初看结果并不乐观，毕竟这种血液检测的准确率高达98%。那让我们用贝叶斯法则来计算看看：

- P(阳性|癌症)P(癌症) = 0.98 * 0.008 = 0.0078
- P(阳性|┐癌症)P(┐癌症) = 0.03 * 0.992 = 0.0298

分类的结果是她不会患有癌症。

如果想得到确切的概率，我们可以使用标准化的方法：

$$P(\text{癌症}|\text{阳性}) = \frac{0.0078}{0.0078 + 0.0298} = 0.21$$

可以看到，血液检测为阳性的人患有这种癌症的概率是21%。



可能你会觉得这并不合情理，毕竟血液检测的准确率有98%，而结果却说Ann很可能并没有这种癌症。事实上，很多人都会有这样的 疑问。

我来说明一下为什么会是这样的结果。很多人只看到了血液检测的准确率是98%，但没有考虑到全美只有0.8%的人患有这种癌症。假设我们给一个有着一百万人口的城市做血液检测，也就是说其中有8,000人患有癌症，992,000人没有。首先，对于那8,000个癌症病人，有7,840个人的血液检测结果会呈阳性，160人会呈阴性。对于992,000人，有962,240人会呈阴性，30,000人呈阳性。将这些数字总结到下表中：

	阳性	阴性
患有癌症	7,840	160
健康	30,000	962,240

Ann的测试结果呈阳性，从上表看阳性中有30,000人其实是健康的，只有7,840人确实患有癌症，所以我们才会认为Ann很有可能是健康的。

> 还是没弄明白？没关系，在接触了更多练习后就会慢慢理解了。

为什么我们需要贝叶斯法则？

首先回顾一下贝叶斯公式：

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

再看看微软购物车的数据：

客户 ID	邮编	是否购买有机食品	是否购买 Sencha 绿茶
1	88005	是	是
2	88001	否	否
3	88001	是	是
4	88005	否	否
5	88003	是	否
6	88005	否	是
7	88005	否	否
8	88001	否	否
9	88005	是	是
10	88003	是	是

比如，我们为居住在邮编为88005地区的客户设置两个事件：买或不买Sencha绿茶，即：

$P(h_1|D)$ = P(买绿茶|88005)

$P(h_2|D)$ = P(¬买绿茶|88005)

你也许会问，这两个概率我们都可以直接从数据中计算得到，为什么还要计算下面这个公式呢？

$$\frac{P(88005|\text{买绿茶})P(\text{买绿茶})}{P(88005)}$$

那是因为在现实问题中要计算P(h|D)往往是很困难的。

以上一节中的医学示例来说，我们想要根据血液测试结果来判断该人是否患有癌症：

P(癌症|阳性) ≈ P(阳性|癌症)P(癌症)

P(┐癌症|阳性) ≈ P(阳性|┐癌症)P(┐癌症)

上面两个式子中，我们更容易计算约等号右边的结果。比如，要计算P(阳性|癌症)，我们可以对一部分已经确诊有癌症的人做血液测试；计算P(阳性|┐癌症)时则可对确定健康的人做测试。而P(癌症)则可直接从政府公布的官方数据中获得，P(┐癌症)更是简单的1 - P(癌症)。

但若要计算P(癌症|阳性）的话就非常有挑战性了，也就是计算出整个人群中血液测试结果呈阳性且确诊为癌症患者的概率。如果采用抽样法，对1000个人进行抽样测试，只有8个人患有这种癌症，这样得出的结果显然不具有代表性，除非进一步加大抽样数量。而贝叶斯法则提供了更为简便的方法。

朴素贝叶斯

很多时候我们会用到不止一个前提条件，比如判断一个人是否会购买Sencha绿茶时可以用到顾客所在地以及是否买过有机食品这两个条件。计算这样的概率时只需将各个条件概率相乘即可：

P(买绿茶|88005 & 买有机食品) = P(88005|买绿茶)P(买有机食品|买绿茶)P(买绿茶) = 0.6 *0.8* 0.5 = 0.24

P(┐买绿茶|88005 & 买有机食品) = P(88005|┐买绿茶)P(买有机食品|┐买绿茶)P(┐买绿茶) = 0.4 *0.25* 0.5 = 0.05

所以得到的结论是居住在88005地区且买过有机食品的客户更有可能购买Sencha绿茶。这样就让我们在智能购物车上显示广告吧！

以下是史提芬贝克对智能购物车的评价：

> 这种购物车的使用体验是：你取走一辆购物车，刷了会员卡，屏幕上会显示一份购物列表，里面的内容都是基于你平时的购物习惯进行推荐的，牛奶、鸡蛋、西葫芦等等。智能系统会提示出购买这些物品的最佳路径。另外，它还允许你修改列表中的商品，比如你可以让它不要再提示菜花和盐焗花生。Accenture的研究表明，人们在购物时会忘记11%的原本打算购买的商品，如果有了这样的智能购物车，就可以省去客户的来回路程，也能为超市增加销量。

i100、i500健康手环

现在我们要为iHealth公司销售健康手环产品，从而和Nike Fuel、Fitbit Flex竞争。iHealth新出产了两件商品：i100和i500：



iHealth 100

能够监测心率，使用GPS导航（从而计算每小时运动公里数等），带WiFi无线，可随时上传数据到iHealth网站上。



iHealth 500

除了提供i100的功能外，还能监测血液含氧量等指标，且提供免费的3G网络连接到iHealth网站。

这些产品通过网络平台销售，所以iHealth雇佣我们开发一套推荐系统。我通过让购买的用户填写调查问卷来收集数据，每个问题都对应一个特征。比如，我们会问客户为什么要开始运动，有三个选项：健康（health）、外表（appearance）、两者皆是（both）；我们会问他目前的运动水平：很少运动（sedentary）、一般（moderate）、经常运动（active）；我们会问他对健身的热情是高（aggressive）还是一般（moderate）；最后，我们会问他是否适应使用高科技产品。整理后的数据如下：

Main Interest	Current Exercise Level	How Motivated	Comfortable with tech. Devices?	Model #
both	sedentary	moderate	yes	i100
both	sedentary	moderate	no	i100
health	sedentary	moderate	yes	i500
appearance	active	moderate	yes	i500
appearance	moderate	aggressive	yes	i500
appearance	moderate	aggressive	no	i100
health	moderate	aggressive	no	i500
both	active	moderate	yes	i100
both	moderate	aggressive	yes	i500
appearance	active	aggressive	yes	i500
both	active	aggressive	no	i500
health	active	moderate	no	i500
health	sedentary	aggressive	yes	i500
appearance	active	moderate	no	i100
health	sedentary	moderate	no	i100

实践

已知一位客户的运动目的是健康、当前水平是中等、热情一般、能适应高科技产品，请用朴素贝叶斯来推荐手环型号。

我们需要计算以下两个概率，并选取较大的结果：

```
P(i100|健康，中等水平、热情一般，适应)
P(i500|健康，中等水平，热情一般，适应)
```

我们先来看第一个概率：

```
P(i100|健康，中等水平、热情一般，适应） = P(健康|i100)P(中等水平|i100)P(热情一般|i100)P(适应|i100)
```

其中：

```
P(健康|i100) = 1/6
P(中等水平|i100) = 1/6
P(热情一般|i100) = 5/6
P(适应|i100) = 2/6
P(i100) = 6/15
```

因此：

```
P(i100|满足条件) = 0.167 * 0.167 * 0.833 * 0.333 * 0.4 = 0.00309
```

再计算另一个模型的概率：

```
P(i500|满足条件) = P(健康|i500)P(中等水平|i500)P(热情一般|i500)P(适应|i500)
    = 4/9 * 3/9 * 3/9 * 6/9 * 9/15
    = 0.444 * 0.333 * 0.333 * 0.667 * 0.6
    = 0.01975
```

使用Python编写朴素贝叶斯分类器

上例的数据格式如下：

```
both    sedentary    moderate    yes    i100
both    sedentary    moderate    no    i100
health    sedentary    moderate    yes    i500
appearance    active    moderate    yes    i500
appearance    moderate    aggressive    yes    i500
appearance    moderate    aggressive    no    i100
health    moderate    aggressive    no    i500
both    active    moderate    yes    i100
both    moderate    aggressive    yes    i500
appearance    active    aggressive    yes    i500
both    active    aggressive    no    i500
health    active    moderate    no    i500
health    sedentary    aggressive    yes    i500
appearance    active    moderate    no    i100
health    sedentary    moderate    no    i100
```

虽然这个这个例子中只有15条数据，但是我们还是保留十折交叉验证的过程，以便用于更大的数据集。十折交叉验证要求数据集等分成10份，这个例子中我们简单地将15条数据全部放到一个桶里，其它桶留空。

朴素贝叶斯分类器包含两个部分：训练和分类。

训练

训练的输出结果应该是：

- 先验概率，如P(i100) = 0.4；
- 条件概率，如P(健康|i100) = 0.167

我们使用如下代码表示先验概率：

```
self.prior = {'i500': 0.6, 'i100': 0.4}
```

条件概率的表示有些复杂，用嵌套的字典来实现：

```
{'i500': {1: {'appearance': 0.3333333333333333, 'health': 0.4444444444444444,
             'both': 0.2222222222222222},
          2: {'active': 0.4444444444444444, 'sedentary': 0.2222222222222222,
             'moderate': 0.3333333333333333},
          3: {'aggressive': 0.6666666666666666, 'moderate': 0.3333333333333333},
          4: {'yes': 0.6666666666666666, 'no': 0.3333333333333333}},
 'i100': {1: {'both': 0.5, 'health': 0.16666666666666666,
             'appearance': 0.3333333333333333},
          2: {'active': 0.3333333333333333, 'sedentary': 0.5,
             'moderate': 0.16666666666666666},
          3: {'aggressive': 0.16666666666666666, 'moderate': 0.8333333333333334},
          4: {'yes': 0.3333333333333333, 'no': 0.6666666666666666}}}
```

1、2、3、4表示第几列，所以第一行可以解释为购买i500的顾客中运动目的是外表的概率是0.333。

首先我们要来进行计数，比如以下几行数据：

```
both     sedentary     moderate     yes     i100
both     sedentary     moderate     no     i100
health     sedentary     moderate     yes     i500
appearance     active     moderate     yes     i500
```

我们用字典来统计每个模型的次数，变量名为classes，逐行扫描后的结果是：

```
# 第一行
{'i100': 1}

# 第二行
{'i100': 2}

# 第三行
{'i500': 1, 'i100': 2}

# 全部
{'i500': 9, 'i100': 6}
```

要获取模型的先验概率，只要将计数结果除以总数就可以了。

计算后验概率也需要计数，变量名为counts。这个字典较为复杂，如扫完第一行第一列的结果是：

```
{'i100': {1: {'both': 1}}
```

处理完所有数据后的计数结果是：

```
{'i500': {1: {'appearance': 3, 'health': 4, 'both': 2},
          2: {'active': 4, 'sedentary': 2, 'moderate': 3},
          3: {'aggressive': 6, 'moderate': 3},
          4: {'yes': 6, 'no': 3}},
 'i100': {1: {'both': 3, 'health': 1, 'appearance': 2},
          2: {'active': 2, 'sedentary': 3, 'moderate': 1},
          3: {'aggressive': 1, 'moderate': 5},
          4: {'yes': 2, 'no': 4}}}
```

计算概率时，只需将计数除以该模型的总数就可以了：

P(外表|i100) = 2 / 6 = 0.333

以下是训练用的Python代码：

```
class Classifier:

    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):
        """bucketPrefix 分桶数据集文件前缀
        testBucketNumber 测试桶编号
        dataFormat 数据格式，形如：attr attr attr attr class
        """

        total = 0
        classes = {}
        counts = {}

        # 从文件中读取数据
        self.format = dataFormat.strip().split('\t')
        self.prior = {}
        self.conditional = {}
        # 遍历十个桶
        for i in range(1, 11):
            # 跳过测试桶
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                f = open(filename)
                lines = f.readlines()
                f.close()
                for line in lines:
                    fields = line.strip().split('\t')
                    ignore = []
                    vector = []
                    for i in range(len(fields)):
                        if self.format[i] == 'num':
                            vector.append(float(fields[i]))
                        elif self.format[i] == 'attr':
                            vector.append(fields[i])
                        elif self.format[i] == 'comment':
                            ignore.append(fields[i])
                        elif self.format[i] == 'class':
                            category = fields[i]
                    # 处理该条记录
                    total += 1
                    classes.setdefault(category, 0)
                    counts.setdefault(category, {})
                    classes[category] += 1
                    # 处理各个属性
                    col = 0
                    for columnValue in vector:
                        col += 1
                        counts[category].setdefault(col, {})
                        counts[category][col].setdefault(columnValue, 0)
                        counts[category][col][columnValue] += 1

        # 计数结束，开始计算概率

        # 计算先验概率P(h)
        for (category, count) in classes.items():
            self.prior[category] = count / total

        # 计算条件概率P(h|D)
        for (category, columns) in counts.items():
              self.conditional.setdefault(category, {})
              for (col, valueCounts) in columns.items():
                  self.conditional[category].setdefault(col, {})
                  for (attrValue, count) in valueCounts.items():
                      self.conditional[category][col][attrValue] = (
                          count / classes[category])
        self.tmp =  counts
```

分类

分类函数会这样使用：

```
c.classify(['health', 'moderate', 'moderate', 'yes'])
```

我们需要计算：

$$h_{MAP} = \arg\max_{h \in H} P(D|h)P(h)$$

```
    def classify(self, itemVector):
        """返回itemVector所属类别"""
        results = []
        for (category, prior) in self.prior.items():
            prob = prior
            col = 1
            for attrValue in itemVector:
                if not attrValue in self.conditional[category][col]:
                    # 属性不存在，返回0概率
                    prob = 0
                else:
                    prob = prob * self.conditional[category][col][attrValue]
                col += 1
            results.append((prob, category))
        # 返回概率最高的结果
        return(max(results)[1])
```

让我们测试一下：

```
>>> c = Classifier('iHealth/i', 10, 'attr\tattr\tattr\tattr\tclass')
>>> c.classify(['health' 'moderate', 'moderate', 'yes'])
i500
```



共和党还是民主党

我们来看另一个数据集——美国国会投票数据，可以从 机器学习仓库 获得。每条记录代表一个选民，第一列是分类名称（democrat, republican），之后是16条法案，用y和n表示该人是否支持。

1. 残疾婴儿法案
2. 用水成本分摊
3. 预算改革
4. 医疗费用
5. 萨瓦尔多援助
6. 校园宗教组织
7. 反卫星武器
8. 尼加拉瓜援助
9. MX导弹
10. 移民法案
11. 合成燃料缩减
12. 教育支出法案
13. 有毒废物基金
14. 犯罪
15. 出口免税
16. 南非出口管控

文件格式如下：

```
democrat    y   n   y   n   n   y   y   y   y   y   n   n   y   n   n   y
democrat    y   y   y   n   n   y   y   y   y   n   n   n   n   n   y   y
republican    y   y   n   y   y   y   n   n   n   y   n   y   y   y   n
```

在调用上一节编写的朴素贝叶斯分类器时使用以下dataFormat参数就可以了：

```
"class\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tattr\tatt
```

十折交叉验证的结果是：

```
               Classified as:
                  democrat   republican
                 +--------+--------+
      democrat   |    111 |     13 |
                 |--------+--------|
    republican   |      9 |     99 |
                 +--------+--------+

 90.517 percent correct
 total of 232 instances
```

看起来不错。

但是，这个方法有一些问题。



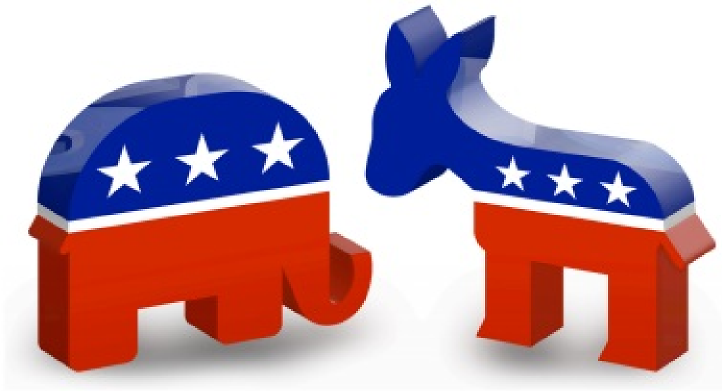
首先我们来模拟一个数据集，其中包含100个民主党派和100个共和党派。下表是他们对四个法案的投票情况：

	网络安全法案	读者隐私	网络营销税负	非法网络传播
共和党	0.99	0.01	0.99	0.5
民主党	0.01	0.99	0.01	1.0

从表中可以看到，共和党人中，99%赞成网络安全法案，只有1%的人赞成读者隐私。下面我们再选取一位议会代表——X先生，对他进行分类：

	网络安全法案	读者隐私	网络营销税负	非法网络传播
共和党	0.99	0.01	0.99	0.5
民主党	0.01	0.99	0.01	1.0
X 先生	否决	赞成	否决	否决

你觉得*X*先生是民主党还是共和党呢？



我会猜测民主党。我们用朴素贝叶斯来进行分类。首先，先验概率P(民主党)和P(共和党)都是0.5，因为样本中两党分别有100人。X先生对网络安全法案投了否决票，并且：

```
P(共和党|C=no) = 0.01
P(民主党|C=no) = 0.99
```

为了表示方便，我们用字母来表示这四个法案：

1. 网络安全：C
2. 读者隐私：R
3. 网销税负：T
4. 非法传播：S

记到表格里就是：

h=	P(h)	P(C=no|h)				P(h|D)
共和党	0.5	0.01				0.005
民主党	0.5	0.99				0.495

我们将X先生对读者隐私和网络营销税负法案的投票记到表格中：

h=	P(h)	P(C=no|h)	P(R=yes|h)	P(T=no|h)		P(h|D)
共和党	0.5	0.01	0.01	0.01		0.0000005
民主党	0.5	0.99	0.99	0.99		0.485

对这些概率进行标准化后可以得到：

$$P(民主党|D) = \frac{0.485}{0.485 + 0.0000005} = \frac{0.485}{0.4850005} = 0.99999$$

也就是说到目前为止我们有99.99%的信心认为X先生是民主党的。

最后，我们将X先生对非法传播法案的投票记入表格：

h=	P(h)	P(C=no|h)	P(R=yes|h)	P(T=no|h)	P(S=no|h)	P(h|D)
共和党	0.5	0.01	0.01	0.01	0.50	2.5E-07
民主党	0.5	0.99	0.99	0.99	0.00	0

你会惊讶地发现，X先生是民主党的可能性从99%降至0了！这是因为我们的数据集中没有一个民主党人对网络非法传播法案投了否决票。

概率估计

使用朴素贝叶斯计算得到的概率其实是真实概率的一种估计，而真实概率是对全量数据做统计得到的。比如说，我们需要对所有人都做血液测试，才能得到健康人返回阴性结果的真实概率。显然，对全量数据做统计是不现实的，所以我们会选取一个样本，如1000人，对他们进行测试并计算概率。大部分情况下，这种估计都是接近于真实概率的。但当真实概率非常小时，这种抽样统计的做法就会有问题了。

比如说，民主党对网络非法传播法案的否决率是0.03，即P(S=no|民主党) = 0.03。如果我们分别选取十个民主党和共和党人，看他们对该法案的投票情况，你觉得得到的概率会是什么？答案很可能是0。

从上一节的例子中也看到了，在朴素贝叶斯中，概率为0的影响是很大的，甚至会不顾其他概率的大小。此外，抽样统计的另一个问题是会低估真实概率。

如何解决

我们计算P(S=no|民主党)的公式是这样的：

$$P(S=no|\text{民主党})=\frac{\text{民主党中对网络非法传播法案投否决票的人数}}{\text{民主党总人数}}$$

为了表示方便，我们采用以下公式：

$$P(x|y)=\frac{n_c}{n}$$

其中，n表示训练集中y类别的记录数；n_c表示y类别中值为x的记录数。

我们的问题是上式中的n_c可能为0，解决方法是将公式变为以下形式：

$$P(x|y)=\frac{n_c+mp}{n+m}$$

这个公式摘自*Tom Mitchell*《机器学习》的第*179*页。

m是一个常数，表示等效样本大小。决定常数m的方法有很多，我们这里使用值的类别来作为m，比如投票有赞成和否决两种类别，所以m就为2。p则是相应的先验概率，比如说赞成和否决的概率分别是0.5，那p就是0.5。

我们回到上面的例子，看看要如何应用这个方法。下表是投票的情况：

共和党

	网络安全	读者隐私	网销税负	非法传播
赞成	99	1	99	50
否决	1	99	1	50

民主党

	网络安全	读者隐私	网销税负	非法传播
赞成	1	99	1	0
否决	99	1	99	100

X先生对网络安全法案投了否决票，我们先来计算共和党对安全法案投否决票的概率。新的公式是：

$$P(x|y)=\frac{n_c+mp}{n+m}$$

其中n是共和党的人数100，n_C是共和党对安全法案投否决票的人数1，m是2，p是0.5，因此：

$$P(C=no|共和党)=\frac{1+2\times0.5}{100+2}=\frac{2}{102}=0.01961$$

再计算民主党：

$$P(C=no|民主党)=\frac{99+2\times0.5}{100+2}=\frac{100}{102}=0.9804$$

代入之前用到的表格中：

h=	P(h)	P(C=no|h)				P(h|D)
共和党	0.5	0.01961				0.0098
民主党	0.5	0.9804				0.4902

对剩余三条法案进行计算，得到的结果是：

h=	P(h)	P(C=no|h)	P(R=yes|h)	P(I=no|h)	P(S=no|h)	P(h|D)
共和党	0.5	0.01961	0.01961	0.01961	0.5	0.000002
民主党	0.5	0.9804	0.9804	0.9804	0.0098	0.004617

因此，X先生的党派是民主党，这和我们的直觉一致。

一点说明

在这个例子中，所有公式里的m都是2，但这并不表示其他数据集也是这样。比如我们之前做的健康手环问卷调查，运动目的有三个选项，是否适应高科技则有两个选项，所以在计算运动目的概率时m=3、p=1/3，代入公式即：

$$P(很少运动|i500)=\frac{n_c+mp}{n+m}=\frac{0+3\times0.333}{100+3}=\frac{1}{103}=0.0097$$

数值型数据

你可能已经注意到，在讨论近邻算法时，我们使用的都是数值型的数据，而在学习朴素贝叶斯算法时，用的是分类型的数据。比如，人们对法案的投票有赞成和否决两类；音乐家可以用他们演奏的乐器来分类等等。这些分类之间是没有距离的，萨克斯手和钢琴家的距离并不

会比鼓手近。而数值型数据则有这种远近之分。

在贝叶斯方法中，我们会对事物进行计数，这种计数则是可以度量的。对于数值型的数据要如何计数呢？通常有两种做法：

方法一：区分类别

我们可以划定几个范围作为分类，如：

- 年龄
  - < 18
  - 18 - 22
  - 23 - 30
  - 31 - 40
  - > 40
- 年薪
  - > $200,000

  - 150,000 - 200,000
  - 100,000 - 150,000
  - 60,000 - 100,000
  - 40,000 - 60,000

划分类别后，进行可以应用朴素贝叶斯方法了。

方法二：高斯分布



> 我想将收入数据进行分类，然后应用朴素贝叶斯算法。
>
> 你的做法已经过时了，我会使用高斯分布和概率密度函数来做。

高斯分布和概率密度函数这两个词听起来很酷，他们的作用也非常大。下面我们将学习如何在朴素贝叶斯算法中使用高斯分布。

首先，我们为健康手环的例子增加一列收入属性：

Main Interest	Current Exercise Level	How Motivated	Comfortable with tech. Devices?	Income (in $1,000)	Model #
both	sedentary	moderate	yes	60	i100
both	sedentary	moderate	no	75	i100
health	sedentary	moderate	yes	90	i500
appearance	active	moderate	yes	125	i500
appearance	moderate	aggressive	yes	100	i500
appearance	moderate	aggressive	no	90	i100
health	moderate	aggressive	no	150	i500
both	active	moderate	yes	85	i100
both	moderate	aggressive	yes	100	i500
appearance	active	aggressive	yes	120	i500
both	active	aggressive	no	95	i500
health	active	moderate	no	90	i500
health	sedentary	aggressive	yes	85	i500
appearance	active	moderate	no	70	i100
health	sedentary	moderate	no	45	i100

我们来看购买i500的用户的收入情况，比如取平均值：

$$mean = \frac{90+125+100+150+100+120+95+90+85}{9} = \frac{955}{9} = 106.111$$

还可以求出标准差：

$$sd = \sqrt{\frac{\sum_i (x_i - \overline{x})^2}{card(x)}}$$

标准差是用来衡量数据的离散程度的，如果所有数据都接近于平均值，那标准差也会比较小。通过公式我们可以计算得到i500用户的收入标准差是20.108。

总体标准差和样本标准差

上面的标准差公式是总体标准差，我们需要对所有的数据进行统计才能得出，比如统计500名学生的成绩，就能计算出总体标准差。但通常我们无法获取总体的数据，比如要统计新墨西哥北部鹿鼠的重量，我们不可能对所有的鹿鼠进行称重，只能选取一部分样本，这时计算得到的就是样本标准差。

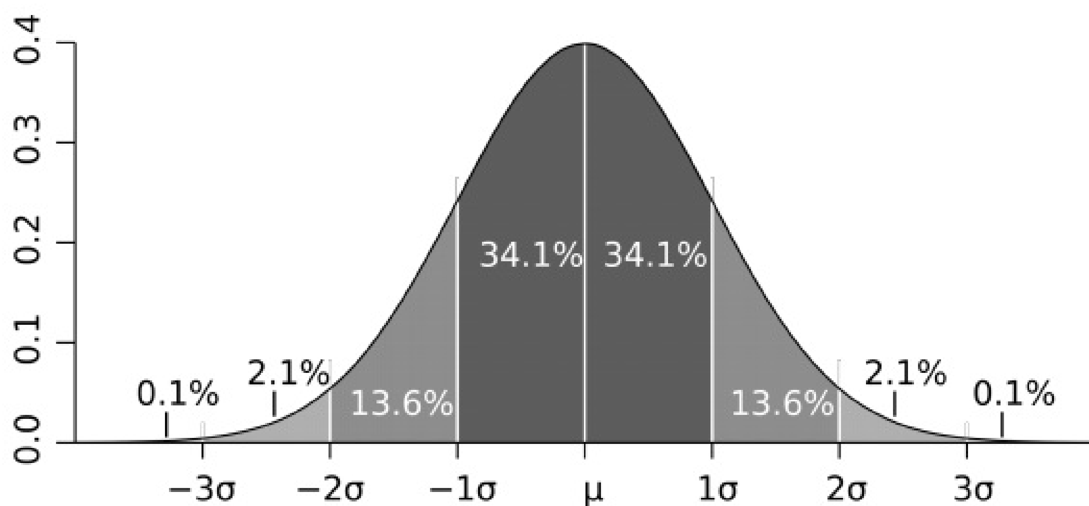


样本标准差的公式是：

$$sd = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{card(x) - 1}}$$

所以计算得到i500用户收入的样本标准差是21.327。下面的内容都会使用样本标准差。

你可能听说过正态分布、钟型曲线、高斯分布等术语，他们指的是同一件事：68%的数据会落在标准差为1的范围内，95%的数据会落在标准差为2的范围内：



在我们的示例中，平均值是106.111，样本标准差是21.327，因此购买i500的用户中有95%的人收入在42,660美元至149,770美元之间。如果我问你P(100k|i500)的概率有多大，你可以回答非常大；如果问你P(20k|i500)的概率有多大，你可以回答基本不可能。

用公式来表示是：

$$P(x_i | y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} e^{\frac{-(x_i - \mu_{ij})^2}{2\sigma_{ij}^2}}$$



> 每次出现这些公式时我都想提醒读者千万不要紧张，其实他们只是看起来比较复杂，只需一步一步拆解开就能理解了。数据挖掘学到后面会遇到各种复杂的公式，千万不要被他们的外表吓到。

让我们逐步拆解这个公式。假设我们要计算P(100k|i500)的概率，即购买i500的用户中收入是100,000美元的概率。之前我们计算过购买i500的用户平均收入以及样本标准差，我们用希腊字母μ（读“谬”）来表示平均值，σ（读“西格玛”）来表示标准差。

$$P(x_i \mid y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} e^{\frac{-(x_i-\mu_{ij})^2}{2\sigma_{ij}^2}}$$

$\mu_{ij} = 106.111$
$\sigma_{ij} = 21.327$
$x_i = 100$

$$P(x_i \mid y_j) = \frac{1}{\sqrt{2\pi}(21.327)} e^{\frac{-(100-106.111)^2}{2(21.327)^2}}$$

$$P(x_i \mid y_j) = \frac{1}{\sqrt{6.283}(21.327)} e^{\frac{-(37.344)}{909.68}}$$

$$P(x_i \mid y_j) = \frac{1}{53.458} e^{-0.0411}$$

e是自然常数，约等于2.718。

$$P(x_i \mid y_j) = \frac{1}{53.458}(2.718)^{-0.0411} = (0.0187)(0.960) = 0.0180$$

练习

下表中列出了加仑公里数为35的车型的马力（HP），现在我想知道同样是35加仑公里的Datsun 280z马力为132的概率。

car	HP
Datsun 210	65
Ford Fiesta	66
VW Jetta	74
Nissan Stanza	88
Ford Escort	65
Triumph tr7 coupe	88
Plymouth Horizon	70
Suburu DL	67

μ_{ij} = 72,875

σ_{ij} = 9.804

x_i =132

$$P(132hp|35mpg)=\frac{1}{\sqrt{2\pi}(9.804)}e^{\frac{-(132-72.875)^2}{2(9.804)^2}}$$

$$=\frac{1}{\sqrt{6.283}(9.804)}e^{\frac{-3495.766}{192.237}}=\frac{1}{24.575}e^{-18.185}$$

$$=0.0407(0.00000001266)$$

$$=0.000000000515$$

结果表明这个概率非常低，而事实上这辆车就是132马力。

代码实现提示

在训练朴素贝叶斯分类器时，可以讲所有属性的平均值和样本标准差计算出来，而分类阶段使用下面这段代码就能实现了：

```
import math

def pdf(mean, ssd, x):
    """概率密度函数，计算P(x|y)"""
    ePart = math.pow(math.e, -(x - mean) ** 2 / (2 * ssd ** 2))
    return (1.0 / (math.sqrt(2 * math.pi) * ssd)) * ePart
```

测试一下：

```
>>> pdf(106.111, 21.327, 100)
0.017953602706962717
>>> pdf(72.875, 9.804, 132)
5.152283971078022e-10
```



> 休息一下吧

使用Python实现

训练阶段

朴素贝叶斯需要用到先验概率和条件概率。让我们回顾一下民主党和共和党的例子：先验概率指的是我们已经掌握的概率，比如美国议会中有233名共和党人，200名民主党人，那共和党人出现的概率就是：

P(共和党) = 233 / 433 = 0.54

我们用P(h)来表示先验概率。而条件概率P(h|D)则表示在已知D的情况下，事件h出现的概率。比如说P(民主党|法案1=yes)。朴素贝叶斯公式中，我们计算的是P(D|h)，如P(法案1=yes|民主党)。

在之前的Python代码中，我们用字典来表示这些概率：

```
{'democrat': {'bill 1': {'yes': 0.333, 'no': 0.667},
              'bill 2': {'yes': 0.778, 'moderate': 0.222}},
 'republican': {'bill 1': {'yes': 0.811, 'no': 0.189},
                'bill 2': {'yes': 0.250, 'no': 0.750}}}
```

所以民主党中对法案1投赞成票的概率是：P(bill 1=yes|民主党) = 0.667。

对于分类型的数据，我们用上面的方法来保存概率，而对连续性的数据，我们要使用概率密度函数，因此需要保存平均值和样本标准差。如：

```
mean = {'democrat': {'age': 57, 'years served': 12},
        'republican': {'age': 53, 'years served': 7}}
ssd = {'democrat': {'age': 7, 'years served': 3},
       'republican': {'age': 5, 'years served': 5}}
```

和之前一样，数据文件中的每一行表示一条记录，不同的特征值使用制表符分隔，比如下面是比马印第安人糖尿病的数据：

```
3	78	50	32	88	31.0	0.248	26	1
4	111	72	47	207	37.1	1.390	56	1
1	189	60	23	846	30.1	0.398	59	1
1	117	88	24	145	34.5	0.403	40	1
3	107	62	13	48	22.9	0.678	23	1
7	81	78	40	48	46.7	0.261	42	0
2	99	70	16	44	20.4	0.235	27	0
5	105	72	29	325	36.9	0.159	28	0
2	142	82	18	64	24.7	0.761	21	0
1	81	72	18	40	26.6	0.283	24	0
0	100	88	60	110	46.8	0.962	31	0
```

前八列是特征，最后一列是分类（1-患病，0-健康）。

我们同样需要一个格式字符串来表示每一行记录：

- `attr` 表示这一列是分类型的特征
- `num` 表示这一列是数值型的特征
- `class` 表示这一列是分类

对于比马数据集，格式化字符串是：

```
"num	num	num	num	num	num	num	num	class"
```

我们需要一个数据结构来存储平均值和样本标准差，看下面这几行数据：

```
3	78	50	32	88	31.0	0.248	26	1
4	111	72	47	207	37.1	1.390	56	1
1	189	60	23	846	30.1	0.398	59	1
2	142	82	18	64	24.7	0.761	21	0
1	81	72	18	40	26.6	0.283	24	0
0	100	88	60	110	46.8	0.962	31	0
```

为计算每一个分类的平均值，我们需要保存合计值，可以用字典来实现：

```
totals = {'1': {1: 8, 2: 378, 3: 182, 4: 102, 5: 1141,
                6: 98.2, 7: 2.036, 8: 141},
          '0': {1: 3, 2: 323, 3: 242, 4: 96, 5: 214,
                6: 98.1, 7: 2.006, 8: 76}}
```

对于分类1，第一列的合计是8（3 + 4 + 1），第二列的合计是378。

对于分类0，第一列的合计是3（2 + 1 + 0），第二列的合计是323，以此类推。

在计算标准差时，我们还需要保留原始的值：

```
numericValues = {'1': {1: [3, 4, 1], 2: [78, 111, 189], ...},
                 '0': {1: [2, 1, 0], 2: [142, 81, 100], ...}}
```

将这些逻辑添加到分类器的 `__init__` 方法中：

```
import math

class Classifier:
    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):
        """bucketPrefix 分桶数据集文件前缀
        testBucketNumber 测试桶编号
        dataFormat 数据格式，形如：attr attr attr attr class
        """
        total = 0
        classes = {}
        # 对分类型数据进行计数
        counts = {}
        # 对数值型数据进行求和
        # 我们会使用下面两个变量来计算每个分类各个特征的平均值和样本标准差
        totals = {}
        numericValues = {}

        # 从文件中读取数据
        self.format = dataFormat.strip().split('\t')
        self.prior = {}
        self.conditional = {}

        # 遍历1-10号桶
        for i in range(1, 11):
            # 判断是否跳过
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                f = open(filename)
                lines = f.readlines()
                f.close()
                for line in lines:
                    fields = line.strip().split('\t')
                    ignore = []
                    vector = []
                    nums = []
                    for i in range(len(fields)):
                        if self.format[i] == 'num':
                            nums.append(float(fields[i]))
                        elif self.format[i] == 'attr':
                            vector.append(fields[i])
                        elif self.format[i] == 'comment':
                            ignore.append(fields[i])
                        elif self.format[i] == 'class':
                            category = fields[i]
                    # 处理这条记录
                    total += 1
                    classes.setdefault(category, 0)
                    counts.setdefault(category, {})
                    totals.setdefault(category, {})
                    numericValues.setdefault(category, {})
                    classes[category] += 1
                    # 处理分类型数据
                    col = 0
                    for columnValue in vector:
                        col += 1
                        counts[category].setdefault(col, {})
                        counts[category][col].setdefault(columnValue, 0)
                        counts[category][col][columnValue] += 1
                    # 处理数值型数据
                    col = 0
                    for columnValue in nums:
                        col += 1
                        totals[category].setdefault(col, 0)
```

```
                    #totals[category][col].setdefault(columnValue, 0)
                    totals[category][col] += columnValue
                    numericValues[category].setdefault(col, [])
                    numericValues[category][col].append(columnValue)

        # 计算先验概率P(h)
        for (category, count) in classes.items():
            self.prior[category] = count / total

        # 计算条件概率P(h|D)
        for (category, columns) in counts.items():
              self.conditional.setdefault(category, {})
              for (col, valueCounts) in columns.items():
                  self.conditional[category].setdefault(col, {})
                  for (attrValue, count) in valueCounts.items():
                      self.conditional[category][col][attrValue] = (
                          count / classes[category])
        self.tmp =  counts

        # 计算平均值和样本标准差
        self.means = {}
        self.ssd = {}
        # 动手实践
```

动手实践[1]

为上述代码实现计算平均值和样本标准差的逻辑，输出的结果如下：

```
>>> c = Classifier('pimaSmall/pimaSmall', 1, 'num\tnum\tnum\tnum\tnum\tnum\tnum\tnum\tcla
>>> c.ssd
{'1': {1: 4.21137914295475, 2: 29.52281872377408, ...},
 '0': {1: 2.54694671925252, 2: 23.454755259159146, ...}}
>>> c.means
{'1': {1: 5.25, 2: 146.05555555555554, ...},
 '0': {1: 2.8867924528301887, 2: 111.90566037735849, ...}}
```

解答

```
        # 计算平均值和样本标准差
        self.means = {}
        self.ssd = {}

        for (category, columns) in totals.items():
            self.means.setdefault(category, {})
            for (col, cTotal) in columns.items():
                self.means[category][col] = cTotal / classes[category]

        for (category, columns) in numericValues.items():
            self.ssd.setdefault(category, {})
            for (col, values) in columns.items():
                SumOfSquareDifferences = 0
                theMean = self.means[category][col]
                for value in values:
                    SumOfSquareDifferences += (value - theMean)**2
                columns[col] = 0
                self.ssd[category][col] = math.sqrt(SumOfSquareDifferences / (classes[cat
```

动手实践[2]

修改分类函数 `classify()`，使其能够使用概率密度函数进行分类。




```
    def classify(self, itemVector, numVector):
        """返回itemVector所属分类"""
        results = []
        sqrt2pi = math.sqrt(2 * math.pi)
        for (category, prior) in self.prior.items():
            prob = prior
            col = 1
            for attrValue in itemVector:
                if not attrValue in self.conditional[category][col]:
                    # 该特征值没有出现过，因此概率给0
                    prob = 0
                else:
                    prob = prob * self.conditional[category][col][attrValue]
                col += 1
            col = 1
            for x in  numVector:
                mean = self.means[category][col]
                ssd = self.ssd[category][col]
                ePart = math.pow(math.e, -(x - mean)**2/(2*ssd**2))
                prob = prob * ((1.0 / (sqrt2pi*ssd)) * ePart)
                col += 1
            results.append((prob, category))
        # 返回概率最高的分类
        #print(results)
        return(max(results)[1])
```

朴素贝叶斯的效果要比近邻算法好吗？

上一章中我们对近邻算法做了统计：

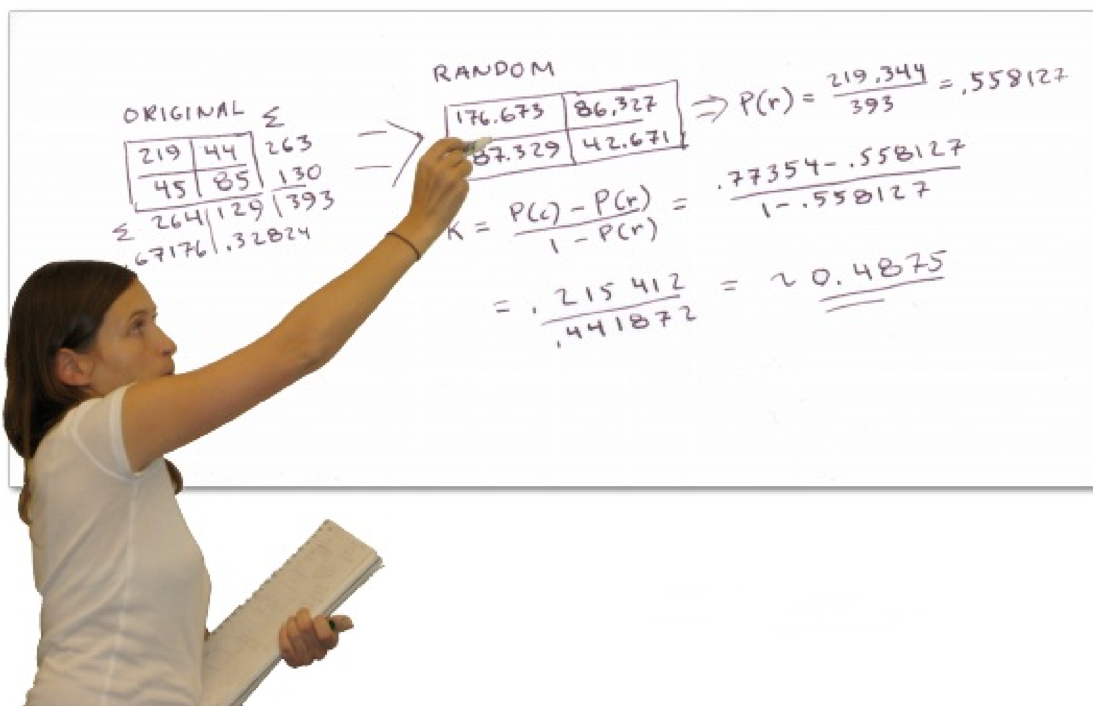
	pimaSmall	pima
k=1	59.00%	71.247%
k=3	61.00%	72.519%

以下是贝叶斯算法的结果：

	pimaSmall	pima
Bayes	72.000%	77.354%



> 哇，看来贝叶斯的效果要比近邻算法来得好呢！
>
> kNN算法中，k=3时的Kappa指标是0.35415，效果一般。那朴素贝叶斯的Kappa指标是多少呢？



> 贝叶斯的Kappa指标是0.4875，符合期望。

因此在这个例子中，朴素贝叶斯的效果要比近邻算法好。

贝叶斯方法的优点：

- 实现简单（只需计数即可）
- 需要的训练集较少
- 运算效率高

贝叶斯方法的主要缺点是无法学习特征之间的相互影响。比如我喜欢奶酪，也喜欢米饭，但是不喜欢两者一起吃。

kNN算法的优点：

- 实现也比较简单
- 不需要按特定形式准备数据
- 需要大量内存保存训练集数据

当我们的训练集较大时，kNN算法是一个不错的选择。这个算法的用途很广，包括推荐系统、蛋白质分析、图片分类等。



> 为什么要叫“朴素贝叶斯”呢？

我们之所以能将多个概率进行相乘是因为这些概率都是具有独立性的。比如说，有一个游戏是同时抛硬币和掷骰子，骰子的点数并不依赖于硬币是正面还是反面，所以在计算联合概率时可以直接相乘。如果我们要计算同时抛出正面（heads）以及掷出6点的概率：

$$P(heads \wedge 6) = P(heads) \times P(6) = 0.5 \times \frac{1}{6} = 0.08333$$

再比如我们有一副扑克牌，保留所有黑色牌（26张），以及红色牌中的人头牌（6张），一共32张。那么，选出一张人头牌（facecard）的概率就是：

$$P(facecard) = \frac{12}{32} = 0.375$$

选出红色牌（red）的概率是：

$$P(red) = \frac{6}{32} = 0.1875$$

那么，选出一张即是红色牌又是人头牌的概率是多少呢？直觉告诉我们不能这样计算：

$$P(red \wedge facecard) = P(red) \times P(facecard) = 0.375 \times 0.185 = 0.0703$$

因为红色牌的概率是0.1875，但这张红色牌100%是人头牌，所以红色人头牌的概率应该是0.1875。

这里不能做乘法就是因为这两个事件不是互相独立的，在选择红色牌时，人头牌的概率就变了，反之亦然。

在现实数据挖掘场景中，这种特征变量之间不独立的情况还是很多的。

- 运动员例子中，身高和体重不是互相独立的，因为高的人体重也会较高。
- 地区邮编、收入、年龄，这些特征也不完全独立，一些地区的房屋都很昂贵，一些地区则只有房车：加州帕罗奥图大多是20岁的年轻人，而亚利桑那州则多是退休人员。
- 在音乐基因工程中，很多特征也是不独立的，如果音乐中有很多变音吉他，那小提琴的概率就降低了。
- 血液检验的结果中，T4和TSH这两个指标通常是呈反比的。

再从你身边找找例子，比如你的车，它的各种特征之间有相关性吗？一部电影呢？亚马逊上的购买记录又如何？

所以，在使用贝叶斯方法时，我们需要互相独立的特征，但现实生活中很难找到这样的应用，因此我们只能假设他们是独立的了！我们完全忽略了这个问题，因此才称为“朴素的”（天真的）贝叶斯方法。不过事实证明朴素贝叶斯的效果还是很不错的。

动手实践

你能将朴素贝叶斯方法应用到其他数据集上吗？比如加仑公里数的例子，kNN算法的正确率是53%，尝试用贝叶斯方法来实现吧。

```
>>> tenfold('mpgData/mpgData', 'class\tattr\tnum\tnum\tnum\tnum\tcomment')
```

第七章：朴素贝叶斯和文本数据

原文：http://guidetodatamining.com/chapter-7/

非结构化文本的分类算法

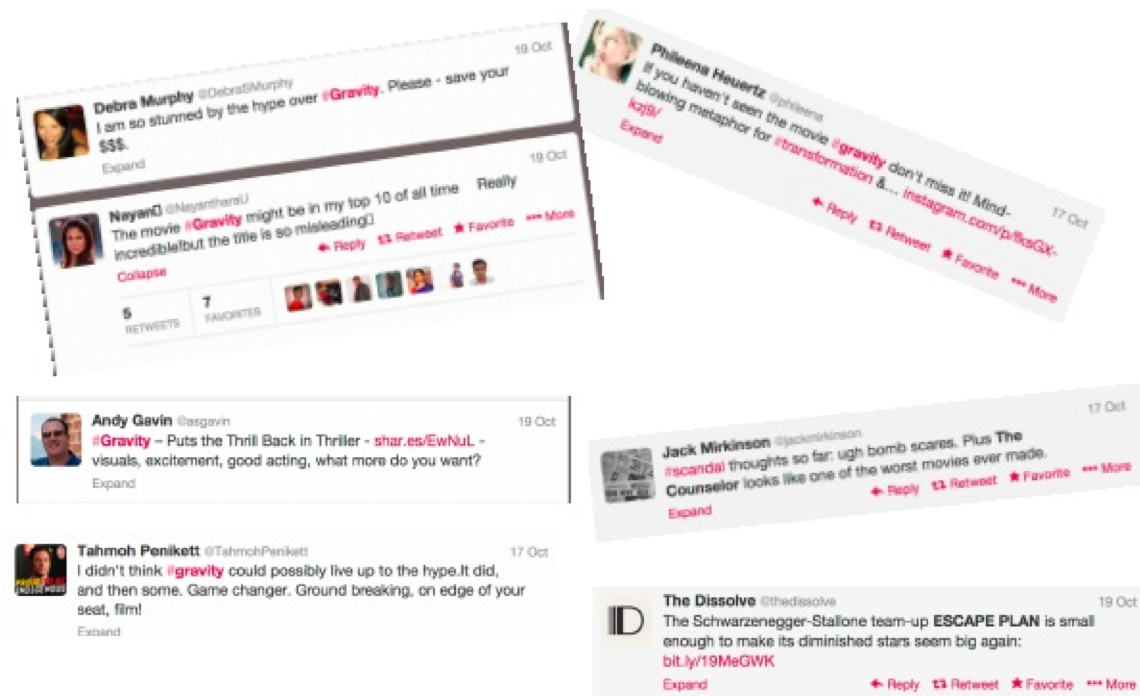
在前几个章节中，我们学习了如何使用人们对物品的评价（五星、顶和踩）来进行推荐；还使用了他们的隐式评价——买过什么，点击过什么；我们利用特征来进行分类，如身高、体重、对法案的投票等。这些数据有一个共性——能用表格来展现：

年龄	葡萄糖水平	血压	是否患有糖尿病
26	78	50	1
56	111	72	1
23	81	78	0

加仑公里数	汽缸数	马力	加速度
30	4	68	19.5
45	4	48	21.7
20	8	130	12

因此这类数据我们称为“结构化数据”——数据集中的每条数据（上表中的一行）由多个特征进行描述（上表中的列）。而非结构化的数据指的是诸如电子邮件文本、推特信息、博客、新闻等。这些数据至少第一眼看起来是无法用一张表格来展现的。

举个例子，我们想从推特信息中获取用户对各种电影的评价：



可以看到，Andy Gavin喜欢看地心引力，因为他的消息中有“不寒而栗”、“演的太棒了”之类的文本。而Debra Murphy则不太喜欢这部电影，因为她说“还是省下看这部电影的钱吧”。如果有人说“我太想看这部电影了，都兴奋坏了！”，我们可以看出她是喜欢这部电影的，即使信息中有“坏”这个字。

我在逛超市时看到一种叫Chobani的酸奶，名字挺有趣的，但真的好吃吗？于是我掏出iPhone，谷歌了一把，看到一篇名为“女人不能只吃面包”的博客：

> 无糖酸奶品评
>
> 你喝过Chobani酸奶吗？如果没有，就赶紧拿起钥匙出门去买吧！虽然它是脱脂原味的，但喝起来和酸奶的口感很像，致使我每次喝都有负罪感，因为这分明就是在喝全脂酸奶啊！原味的感觉很酸很够味，你也可以尝试一下蜂蜜口味的。我承认，虽然我在减肥期间不该吃蜂蜜的，但如果我有一天心情很糟想吃甜食，我就会在原味酸奶里舀一勺蜂蜜，太值得了！至于那些水果味的，应该都有糖分在里面，但其实酸奶本身就已经很美味了，水果只是点缀。如果你家附近没有Chobani，也可以试试Fage，同样好吃。
>
> 虽然需要花上一美元不到，而且还会增加20卡路里，但还是很值得的，毕竟我已经一下午没吃东西了！
>
> *http://womandoesnotliveonbreadalone.blogspot.com/2009/03/sugar-free-yogurt-reviews.html*

这是一篇正面评价吗？从第二句就可以看出，作者非常鼓励我去买。她还用了“够味”、“美味”等词汇，这些都是正面的评价。所以，让我先去吃会儿……



自动判别文本中的感情色彩



> 约翰，这条推文应该是称赞地心引力的！

假设我们要构建一个自动判别文本感情色彩的系统，它有什么作用呢？比如说有家公司是售卖健康检测设备的，他们想要知道人们对这款产品的反响如何。他们投放了很多广告，顾客是喜欢（我好想买一台）还是讨厌（看起来很糟糕）呢？再比如苹果公司召开了一次新闻发布会，讨论iPhone现有的问题，结果是正面的还是负面的呢？一位参议会议员对某个法案做了一次公开演讲，那些政治评论家的反应如何？看来这个系统还是有些作用的。

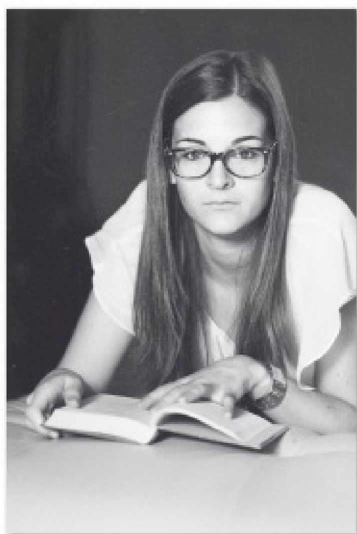


> 那要怎样构建一套这样的系统呢？

假设我要从文本中区分顾客对某些食品的喜好，可能就会列出一些表达喜欢的词语，以及表达厌恶的词：

- 表达喜欢的词：美味、好吃、不错、喜欢、可口
- 表达厌恶的词：糟糕、难吃、不好、讨厌、恶心

比如我们想知道某篇评论对Chobani酸奶的评价是正面的还是负面的，我们可以去统计评论中表达喜欢和厌恶的词的数量，看哪种类型出现的频率高。这种方法也可以应用到其他分类中，比如判断某个人是否支持堕胎，如果他的言论中经常出现“未出生的小孩”，那他很可能是反堕胎的；如果言论中出现“胎儿”这个词比较多，那有可能是支持堕胎的。其实，用词语出现的数量来进行分类还是很容易想到的。



我们可以使用朴素贝叶斯算法来进行分类，而不是一般的计数。先来回忆一下公式：

$$h_{MAP} = \arg\max_{h \in H} P(D|h)P(h)$$

argmax表示选取概率最大的分类；h∈H表示计算每个事件的概率；P(D|h)表示在给定h的条件下，D发生的概率（如给定某类文章，这类文章中特定单词出现的概率）；P(h)则指事件h发生的概率。

我们的训练集是一组文本，又称为语料库。每个文本（即每条记录）是一则140字左右的推文，并被标记为喜欢和讨厌两类。P(h)表示的就是喜欢和讨厌出现的概率。我们的训练集中有1000条记录，喜欢和讨厌各有500条，因此它们的概率是：

```
P(喜欢) = 0.5
P(讨厌) = 0.5
```



> 当我们使用已经标记好分类的数据集进行训练时，这种类型的机器学习称为“监督式学习”。文本分类就是监督式学习的一种。
>
> 如果训练集没有标好分类，那就称为“非监督式学习”，聚类就是一种非监督式学习，我们将在下一章讲解。
>
> 还有一些算法结合了监督式和非监督式，通常是在初始化阶段使用分类好的数据，之后再使用未分类的数据进行学习。

让我们回到上面的公式，首先来看P(D|h)要如何计算——在正面评价中，单词D出现的概率。比如说“Puts the Thrill back in Trhiller”这句话，我们可以统计所有表达“喜欢”的文章中第一个单词是“Puts”的概率，第二个单词是“the”的概率，以此类推。接着我们再计算表达“讨厌”的文章中第一个单词是“Puts”的概率，第二个单词是“the”的概率等等。



> 谷歌曾统计过英语中大约有一百万的词汇，如果一条推文中有14个单词，那我们就需要计算$1{,}000{,}000^{14}$个概率了，显然是不现实的。

的确，这种方法并不可行。我们可以简化一下，不考虑文本中单词的顺序，仅统计表达“喜欢”的文章中某个单词出现的概率。以下是统计方法。

训练阶段

首先，我们统计所有文本中一共出现了多少个不同的单词，记作“|Vocabulary|”（总词汇表）。对于每个单词w_k，我们将计算$P(w_k|h_i)$，每个h_i（喜欢和讨厌两种）的计算步骤如下：

1. 将该分类下的所有文章合并到一起；
2. 统计每个单词出现的数量，记为n；
3. 对于总词汇表中的单词w_k，统计他们在本类文章中出现的次数n_k：
4. 最后应用下方的公式：

$$P(w_k|h_i)=\frac{n_k+1}{n+|Vocabulary|}$$

使用朴素贝叶斯进行分类

分类阶段比较简单，直接应用贝叶斯公式就可以了，让我们试试吧！



通过训练，我们得到以下概率结果：

word	P(word|like)	P(word|dislike)
am	0.007	0.009
by	0.012	0.012
good	0.002	0.0005
gravity	0.00001	0.00001
great	0.003	0.0007
hype	0.0007	0.002
I	0.01	0.01
over	0.005	0.0047
stunned	0.0009	0.002
the	0.047	0.0465

比如下面这句话，要如何判断它是正面还是负面的呢？

> I am stunned by the hype over gravity.

我们需要计算的是下面两个概率，并选取较高的结果：

```
P(like)×P(I|like)×P(am|like)×P(stunned|like)×...
P(dislike)×P(I|dislike)×P(am|dislike)×P(stunned|dislike)×...
```

word	P(word|like)	P(word|dislike)
	P(like) = 0.5	P(dislike) =0.05
I	0.01	0.01
am	0.007	0.009
stunned	0.0009	0.002
by	0.012	0.012
the	0.047	0.0465
hype	0.0007	0.002
over	0.005	0.0047
gravity	0.00001	0.00001
Π	6.22E-22	4.72E-21

因此分类的结果是“讨厌”。

提示 结果中的6.22E-22是科学计数法，等价于6.22×10^{-22}。



> 哇，这个概率也太小了吧！
>
> 是的，如果文本中有100个单词，那乘出来的概率就会更小。
>
> 但是Python不能处理那么小的小数，最后都会变成零的。
>
> 没错，因此我们要用对数来算——将每个概率的对数相加！

假设一个包含100字的文本中，每个单词的概率是0.0001，那么计算结果是：

```
>>> 0.0001 ** 100
0.0
```

如果我们用对数相加来运算的话：

```
>>> import math
>>> p = 0
>>> for i in range(100):
...     p += math.log(0.0001)
...
>>> p
-921.034037197617
```

提示

- $b^n = x$ 可以转换为 $\log_b x = n$
- $\log_{10}(ab) = \log_{10}(a) + \log_{10}(b)$

新闻组语料库

我们下面要处理的数据集是新闻，这些新闻可以分为不同的新闻组，我们会构造一个分类器来判断某则新闻是属于哪个新闻组的：

comp.graphics	misc.forsale	soc.religion.christian	alt.atheism
comp.os.ms-windows-misc	rec.autos	talk.politics.guns	sci.space
comp.sys.ibm.pc.hardware	rec.motorcycles	talk.politics.mideast	sci.crypt
comp.sys.mac.hardware	rec.sport.baseball	talk.politics.misc	sci.electronics
comp.windows.x	rec.sport.hockey	talk.religion.misc	sci.med

比如下面这则新闻是属于rec.motorcycles组的：

From: essbaum@rchland.vnet.ibm.com
(Alexander Essbaum)
Subject: Re: Mail order response time
Disclaimer: This posting represents the poster's views, not necessarily those of IBM
Nntp-Posting-Host: relva.rchland.ibm.com
Organization: IBM Rochester
Lines: 18
> I have ordered many times from Competition
> accesories and ussually get 2-3 day delivery.

ordered 2 fork seals and 2 guide bushings from CA for my FZR. two weeks later get 2 fork seals and 1 guide bushing. call CA and ask for remaining *guide* bushing and order 2 *slide* bushings (explain on the phone which bushings are which; the guy seemed to understand). two weeks later get 2 guide bushings.

sigh

how much you wanna bet that once i get ALL the parts and take the fork apart that some parts won't fit?

注意到这则新闻中还有一些拼写错误（如accesories、ussually等），这对分类器是一个不小的挑战。

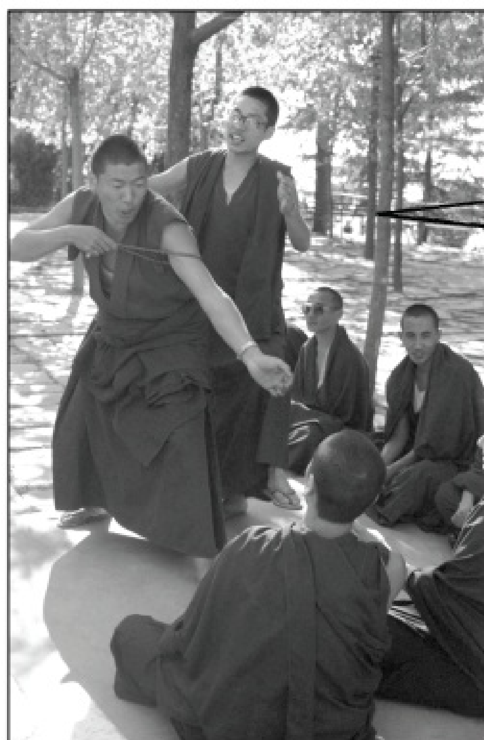
这些数据集都来自 http://qwone.com/~jason/20Newsgroups/ （我们使用的是20news-bydate数据集），你也可以从 这里 获得。这个数据集包含18,846个文档，并将训练集（60%）和测试集放在了不同的目录中，每个子目录都是一个新闻组，目录中的文件即新闻文本。

把不要的东西丢掉！

比如我们要对下面这篇新闻做分类：

> I am looking at buying a Dual Sport type motorcycle. This is my first
> cycle as well. I am interested in any experiences people have with
> the following motorcycles, good or bad.
>
> Honda XR250L
> Suzuki DR350S
> Suzuki DR250ES
> Yamaha XT350
>
> Most XXX vs. YYY articles I have seen in magazines pit the Honda XR650L
> against another cycle, and the 650 always comes out shining. Is it safe
> to assume that the 250 would be of equal quality ?

让我们看看哪些单词是比较重要的：



(helpful - 重要，not helpful - 不重要）

如果我们将英语中最常用的200个单词剔除掉，这篇新闻就成了这样：

I am looking at buying a Dual Sport type motorcycle. This is my first cycle as well. I am interested in any experiences people have with the following motorcycles, good or bad.

Honda XR250L
Suzuki DR350S
Suzuki DR250ES
Yamaha XT350

Most XXX vs. YYY articles I have seen in magazines pit the Honda XR650L against another cycle, and the 650 always comes out shining. Is it safe to assume that the 250 would be of equal quality ?

去除掉这些单词后，新闻就只剩下一半大小了。而且，这些单词看上去并不会对分类结果产生影响。H.P. Luhn在他的论文中说“这些组成语法结构的单词是没有意义的，反而会产生很多噪音”。也就是说，将这些“噪音”单词去除后是会提升分类正确率的。我们将这些单词称为“停词”，有专门的停词表可供使用。去除这些词的理由是：

1. 能够减少需要处理的数据量；
2. 这些词的存在会对分类效果产生负面影响。

常用词和停词

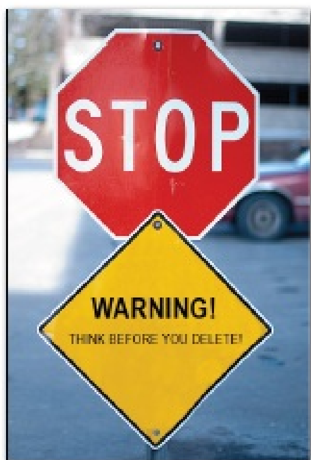
虽然像the、a这种单词的确没有意义，但有些常用词如work、write、school等在某些场合下还是有作用的，如果将他们也列进停词表里可能会有问题。



> 年轻人，那些常用词是不能随便丢弃的！

因此在定制停词表时还是需要做些考虑的。比如要判别阿拉伯语文档是在哪个地区书写的，可以只看文章中最常出现的词（和上面的方式相反）。如果你有兴趣，可以到我的 个人网站 上看看这篇论文。而在分析聊天记录时，强奸犯会使用更多I、me、you这样的词汇，如果在

分析前将这些单词去除了，效果就会变差。



> 不要盲目地使用停词表！

编写Python代码

首先让我们实现朴素贝叶斯分类器的训练部分。训练集的格式是这样的：

```
20news-bydate-train
      alt.atheism
            text file 1 for alt.atheism
            text file 2
            …
            text file n
      comp.graphics
            text file 1 for comp.graphics
            ...
```

最上层的目录是训练集（20news-bydate-train），其下的子目录代表不同的新闻组（如alt.atheism），子目录中有多个文本文件，即新闻内容。测试集的目录结构也是相同的。因此，分类器的初始化代码要完成以下工作：

1. 读取停词列表；
2. 获取训练集中各目录（分类）的名称；
3. 对于各个分类，调用train方法，统计单词出现的次数；
4. 计算下面的公式：

$$P(w_k \mid h_i) = \frac{n_k + 1}{n + |Vocabulary|}$$

```
from __future__ import print_function
import os, codecs, math

class BayesText:
```

```
    def __init__(self, trainingdir, stopwordlist):
        """朴素贝叶斯分类器
        trainingdir 训练集目录，子目录是分类，子目录中包含若干文本
        stopwordlist 停词列表（一行一个）
        """
        self.vocabulary = {}
        self.prob = {}
        self.totals = {}
        self.stopwords = {}
        f = open(stopwordlist)
        for line in f:
            self.stopwords[line.strip()] = 1
        f.close()
        categories = os.listdir(trainingdir)
        # 将不是目录的元素过滤掉
        self.categories = [filename for filename in categories
                           if os.path.isdir(trainingdir + filename)]
        print("Counting ...")
        for category in self.categories:
            print('    ' + category)
            (self.prob[category],
             self.totals[category]) = self.train(trainingdir, category)
        # 删除出现次数小于3次的单词
        toDelete = []
        for word in self.vocabulary:
            if self.vocabulary[word] < 3:
                # 遍历列表时不能删除元素，因此做一个标记
                toDelete.append(word)
        # 删除
        for word in toDelete:
            del self.vocabulary[word]
        # 计算概率
        vocabLength = len(self.vocabulary)
        print("Computing probabilities:")
        for category in self.categories:
            print('    ' + category)
            denominator = self.totals[category] + vocabLength
            for word in self.vocabulary:
                if word in self.prob[category]:
                    count = self.prob[category][word]
                else:
                    count = 1
                self.prob[category][word] = (float(count + 1)
                                             / denominator)
        print ("DONE TRAINING\n\n")


    def train(self, trainingdir, category):
        """计算分类下各单词出现的次数"""
        currentdir = trainingdir + category
        files = os.listdir(currentdir)
        counts = {}
        total = 0
        for file in files:
            #print(currentdir + '/' + file)
            f = codecs.open(currentdir + '/' + file, 'r', 'iso8859-1')
            for line in f:
                tokens = line.split()
                for token in tokens:
                    # 删除标点符号，并将单词转换为小写
                    token = token.strip('\'".,?:-')
                    token = token.lower()
                    if token != '' and not token in self.stopwords:
                        self.vocabulary.setdefault(token, 0)
                        self.vocabulary[token] += 1
                        counts.setdefault(token, 0)
                        counts[token] += 1
                        total += 1
            f.close()
        return(counts, total)
```

训练结果存储在一个名为prop的字典里，字典的键是分类，值是另一个字典——键是单词，值是概率。

```
bT = BayesText(trainingDir, stoplistfile)
>>>bT.prob["rec.motorcycles"]["god"]
0.00013035445075435553
>>>bT.prob["soc.religion.christian"]["god"]
0.004258192391884386
>>>bT.prob["rec.motorcycles"]["the"]
0.028422937849264914
>>>bT.prob["soc.religion.christian"]["the"]
0.039953678998362795
```

god这个词在rec.motorcycles新闻组中出现的概率是0.00013，而在soc.religion.christian新闻组中出现的概率是0.00424。

训练阶段的另一个产物是分类列表：

```
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', ...]
```



训练结束了，下面让我们开始进行文本分类吧。

请尝试编写一个分类器，达成以下效果：

```
>>> bT.classify("20news-bydate-test/rec.motorcycles/104673")
'rec.motorcycles'
>>> bT.classify("20news-bydate-test/sci.med/59246")
'sci.med'
>>> bT.classify("20news-bydate-test/soc.religion.christian/21424")
'soc.religion.christian'
```




```
    def classify(self, filename):
        results = {}
        for category in self.categories:
            results[category] = 0
        f = codecs.open(filename, 'r', 'iso8859-1')
        for line in f:
            tokens = line.split()
            for token in tokens:
                #print(token)
                token = token.strip('\'".,?:-').lower()
                if token in self.vocabulary:
                    for category in self.categories:
                        if self.prob[category][token] == 0:
                            print("%s %s" % (category, token))
                        results[category] += math.log(
                            self.prob[category][token])
        f.close()
        results = list(results.items())
        results.sort(key=lambda tuple: tuple[1], reverse = True)
        # 如果要调试，可以打印出整个列表。
        return results[0][0]
```

最后我们编写一个函数对测试集中的所有文档进行分类，并计算准确率：

```
    def testCategory(self, directory, category):
        files = os.listdir(directory)
        total = 0
        correct = 0
        for file in files:
            total += 1
            result = self.classify(directory + file)
            if result == category:
                correct += 1
        return (correct, total)

    def test(self, testdir):
        """测试集的目录结构和训练集相同"""
        categories = os.listdir(testdir)
        # 过滤掉不是目录的元素
        categories = [filename for filename in categories if
                      os.path.isdir(testdir + filename)]
        correct = 0
        total = 0
        for category in categories:
            print(".", end="")
            (catCorrect, catTotal) = self.testCategory(
                testdir + category + '/', category)
            correct += catCorrect
            total += catTotal
        print("\n\nAccuracy is  %f%%  (%i test instances)" %
              ((float(correct) / total) * 100, total))
```

在不使用停词列表的情况下，这个分类器的效果是：

```
DONE TRAINING

Running Test ...

....................

Accuracy is  77.774827%  (7532 test instances)
```



> 准确率77.77%，看起来很不错。如果用了停词列表效果会如何呢？
>
> 那让我们来测试一下吧！

请自行到网络上查找一些停词列表，并填写以下表格：

stop list size	accuracy
0	77.774827
list 1	
list 2	

我找到了两个停词列表，分别是包含25个词 和174个词 的列表，结果如下：

stop list size	accuracy
0	77.774827%
25 word list	78.757302%
174 word list	79.938927%

看来第二个停词列表能提升2%的效果，你的结果如何？

朴素贝叶斯与情感分析

情感分析的目的是判断作者的态度或意见：



情感分析的例子之一是判断一篇评论是正面的还是反面的，我们可以用朴素贝叶斯算法来实现。我们可以用Pang&Lee 2004的影评数据来测试，这份数据集包含1000个正面和1000个负面的评价，以下是一些示例：

> 本月第二部连环杀人犯电影实在太糟糕了！虽然开头的故事情节和场景布置还可以，但后面就……
>
> 当我听说罗密欧与朱丽叶又出了一部改编电影后，心想莎士比亚的经典又要被糟蹋了。不过我错了，Baz Luhrman导演的水平还是高的……

你可以从 http://www.cs.cornell.edu/People/pabo/movie-review-data/ 上下载这个数据集，并整理成以下形式：

```
review_polarity_buckets
    txt_sentoken
        neg
            0
                files in fold 0
            1
                files in fold 1
            ...
            9
                files in fold 9
        pos
            0
                files in fold 0
            ...
```

你也可以从这里下载整理好的数据。

动手实践

你可以为上文的朴素贝叶斯分类器增加十折交叉验证的逻辑吗？它的输出结果应该是如下形式：

```
        Classified as:
           neg    pos
         +------+------+
neg      |    1 |    2 |
pos      |    3 |    4 |
         +------+------+
12.345 percent correct
total of 2000 instances
```

另外，请计算Kappa指标。

再次声明：只看不练是不行的，就好比你不可能通过阅读乐谱就学会弹奏钢琴。



解答

这是我得到的结果：

```
       Classified as:
          neg    pos
        +------+------+
 neg    | 845  | 155  |
 pos    | 222  | 778  |
        +------+------+

81.150 percent correct
total of 2000 instances
```

Kappa指标则是：

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)} = \frac{.8115 - 0.5}{1 - 0.5} = \frac{.3115}{.5} = 0.623$$

所以我们的分类算法效果是不错的。

代码链接

第八章：聚类

原文：http://guidetodatamining.com/chapter-8/

前几章我们学习了如何构建分类系统，使用的是已经标记好类别的数据集进行训练：

sport	Height	Weight
basketball	72	162
gymnastics	54	66
track	63	106
basketball	78	204

plasma glucose	diastolic BP	BMI	diabetes?
99	52	24.6	0
83	58	34.4	0
139	80	31.6	1

训练完成后我们就可以用来预测了：这个人看起来像是篮球运动员，那个人可能是练体操的；这个人三年内不会患有糖尿病。

可以看到，分类器在训练阶段就已经知道各个类别的名称了。那如果我们不知道呢？如何构建一个能够自动对数据进行分组的系统？比如有1000人，每人有20个特征，我想把这些人分为若干个组。



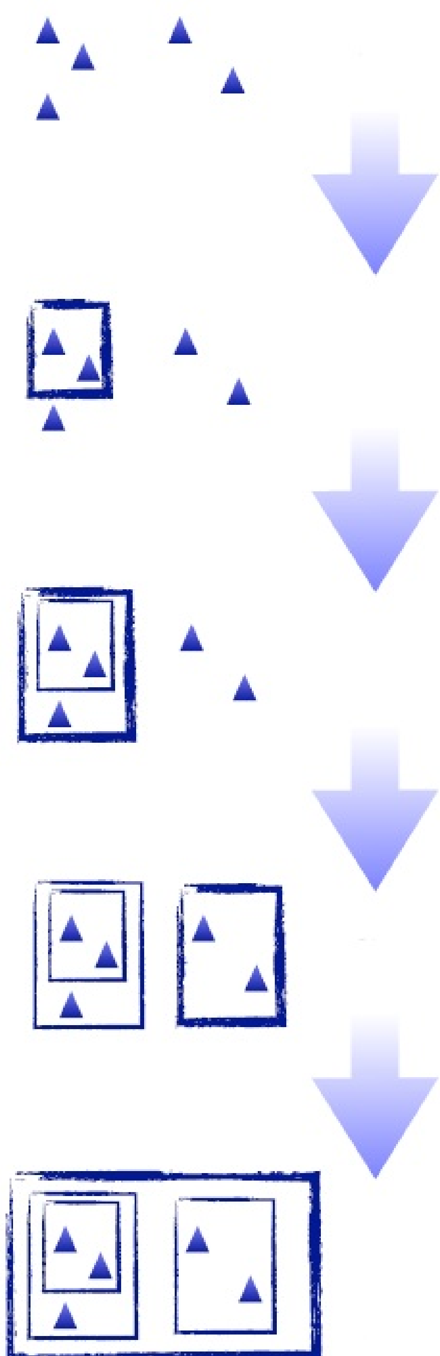
这个过程叫做聚类：通过物品特征来计算距离，并自动分类到不同的群集或组中。有两种聚类算法比较常用：

k-means聚类算法

我们会事先告诉这个算法要将数据分成几个组，比如“请把这1000个人分成5个组”，“将这些网页分成15个组”。这种方法就叫k-means，我们会在后面的章节讨论。

层次聚类法

对于层次聚类法，我们不需要预先指定分类的数量，这个算方法会将每条数据都当作是一个分类，每次迭代的时候合并距离最近的两个分类，直到剩下一个分类为止。因此聚类的结果是：顶层有一个大分类，这个分类下有两个子分类，每个子分类下又有两个子分类，依此类推，层次聚类也因此得命。



在合并的时候我们会计算两个分类之间的距离，可以采用不同的方法。如下图中的A、B、C三个分类，我们应该将哪两个分类合并起来呢？



单链聚类

在单链聚类中，分类之间的距离由两个分类相距最近的两个元素决定。如上图中分类A和分类B的距离由A1和B1的距离决定，因为这个距离小于A1到B2、A2到B1的距离。这样一来我们会将A和B进行合并。

全链聚类

在全链聚类中，分类之间的距离由两个分类相距最远的两个元素决定。因此上图中分类A和B的距离是A2到B2的距离，最后会将分类B和C进行合并。

平均链接聚类

在这种聚类方法中，我们通过计算分类之间两两元素的平均距离来判断分类之间的距离，因此上图中会将分类B和C进行合并。



> 下面让我们用单链聚类法举个例子吧！

我们来用狗的高度和重量来进行聚类：

breed	height (inches)	weight (pounds)
Border Collie	20	45
Boston Terrier	16	20
Brittany Spaniel	18	35
Bullmastiff	27	120
Chihuahua	8	8
German Shepherd	25	78
Golden Retriever	23	70
Great Dane	32	160
Portuguese Water Dog	21	50
Standard Poodle	19	65
Yorkshire Terrier	6	7



> 在计算距离前我们是不是忘了做件事？



> 标准化！我们先将这些数据转换为修正的标准分。

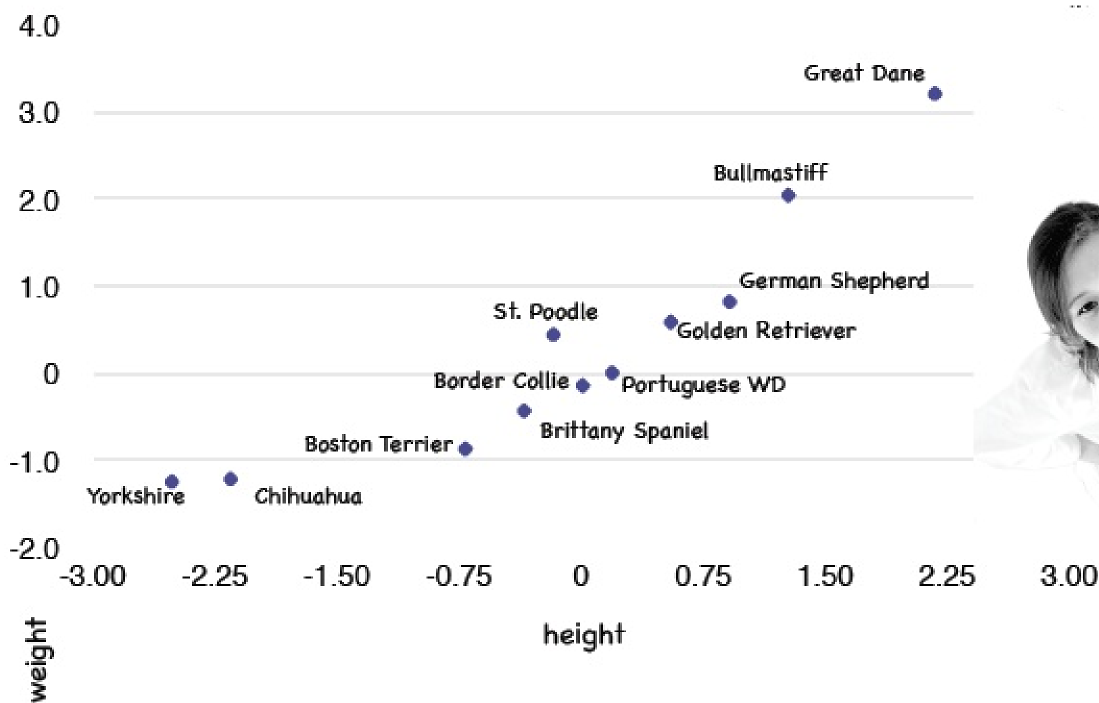
breed	height	weight
Border Collie	0	-0.1455
Boston Terrier	-0.7213	-0.873
Brittany Spaniel	-0.3607	-0.4365
Bullmastiff	1.2623	2.03704
Chihuahua	-2.1639	-1.2222
German Shepherd	0.9016	0.81481
Golden Retriever	0.541	0.58201
Great Dane	2.16393	3.20106
Portuguese Water Dog	0.1803	0
Standard Poodle	-0.1803	0.43651
Yorkshire Terrier	-2.525	-1.25132



然后我们计算欧几里德距离，图中高亮了一些最短距离：

	BT	BS	B	C	GS	GR	GD	PWD	SP	YT
Border Collie	1.024	0.463	2.521	2.417	1.317	0.907	3.985	0.232	0.609	2.756
Boston Terrier		0.566	3.522	1.484	2.342	1.926	4.992	1.255	1.417	1.843
Brittany Spaniel			2.959	1.967	1.777	1.360	4.428	0.695	0.891	2.312
Bullmastiff				4.729	1.274	1.624	1.472	2.307	2.155	5.015
Chihuahua					3.681	3.251	6.188	2.644	2.586	0.362
German Shphrd						0.429	2.700	1.088	1.146	4.001
Golden Retriever							3.081	0.685	0.736	3,572
Great Dane								3.766	3.625	6.466
Portuguese WD									0.566	2.980
Standard Poodle										2.889

根据下面的图表，你能看出哪两个品种的距离最近吗？



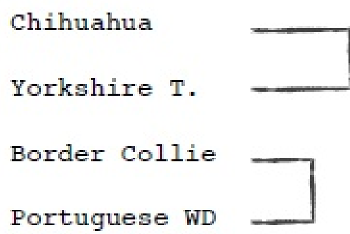
> 如果你看出是Border Collie和Portuguese Water Dog最近，那就对了！

计算过程

第一步：我们找到距离最近的两个元素，对他们进行聚类：

```
Border Collie  ─┐
                │
Portuguese WD  ─┘
```

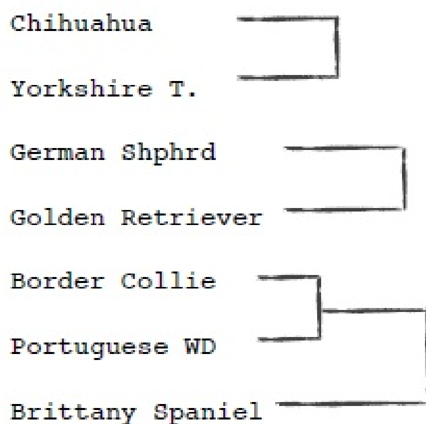
第二步：再找出距离最近的两个元素，进行聚类：



第三步：继续重复上面的步骤：



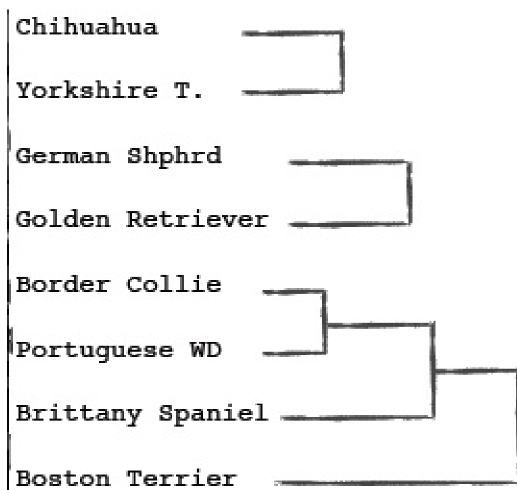
第四步：继续查找距离最近的元素，发现Border Collie已经属于一个分类的，因此进行如下图所示的合并：



> 这叫树状图，可以用来表示聚类。

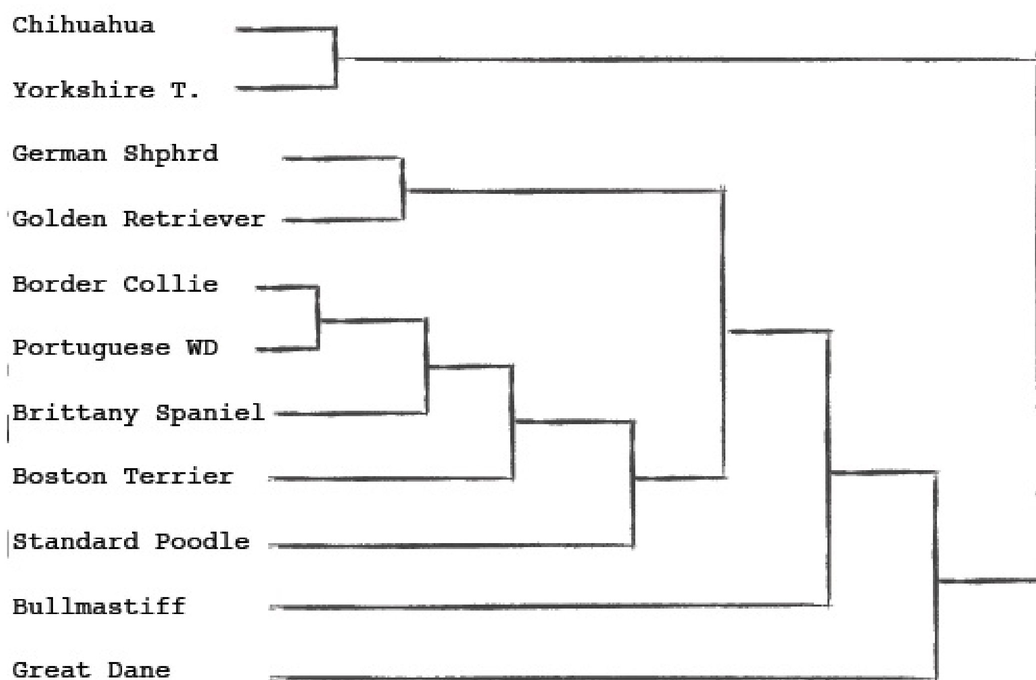
动手实践

你能在下图的基础上继续完成聚类吗？


Chihuahua
Yorkshire T.
German Shphrd
Golden Retriever
Border Collie
Portuguese WD
Brittany Spaniel
Boston Terrier



解答


Chihuahua
Yorkshire T.
German Shphrd
Golden Retriever
Border Collie
Portuguese WD
Brittany Spaniel
Boston Terrier
Standard Poodle
Bullmastiff
Great Dane

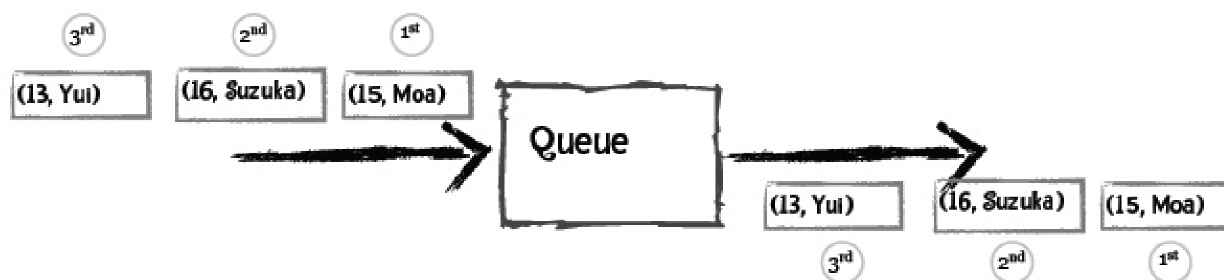


编写层次聚类算法

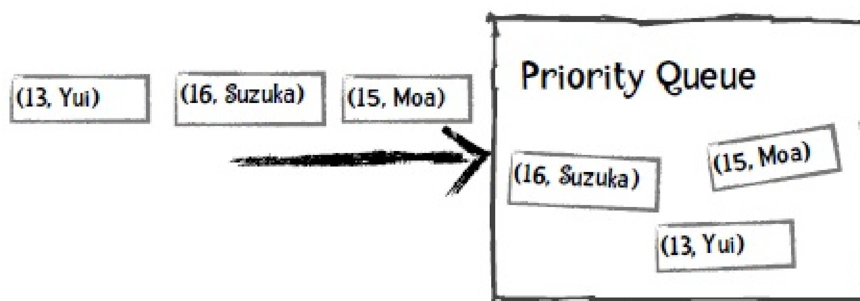


> 我们可以使用优先队列来实现这个聚类算法。
>
> 什么是优先队列呢？

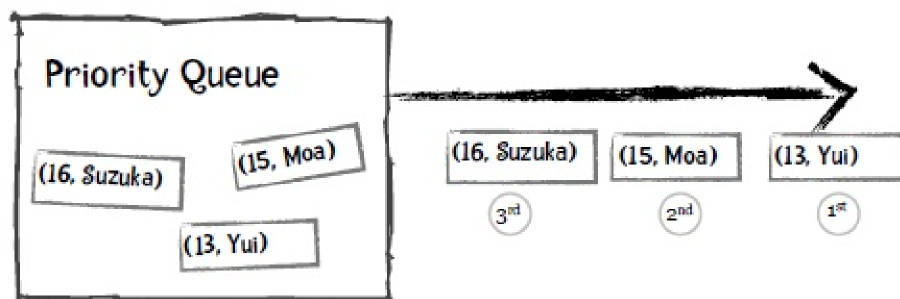
普通的队列有“先进先出”的规则，比如向队列先后添加Moa、Suzuka、Yui，取出时得到的也是Moa、Suzuka、Yui：



而对于优先队列，每个元素都可以附加一个优先级，从队列中取出时会得到优先级最高的元素。比如说，我们定义年龄越小优先级越高，以下是插入过程：



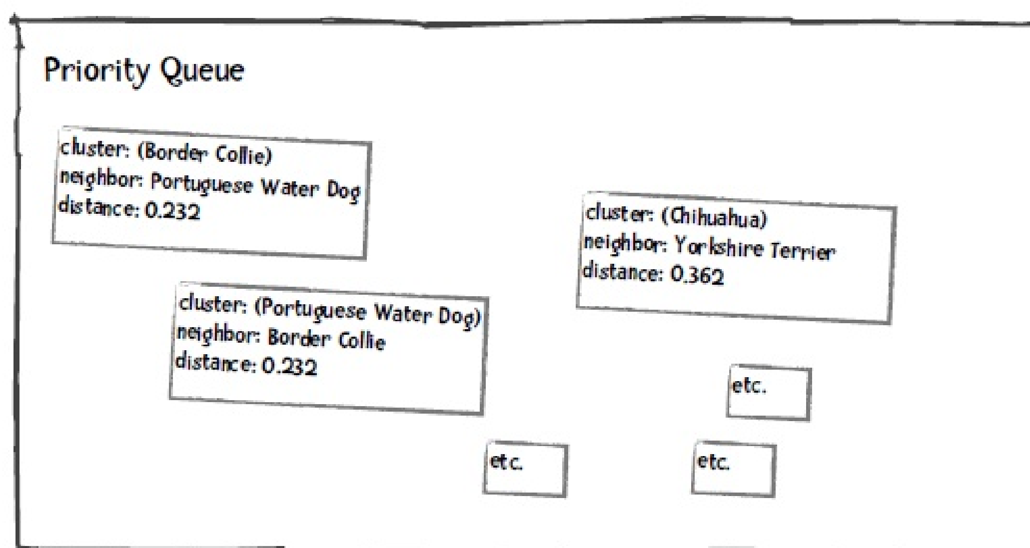
取出的第一个元素是Yui，因为她的年龄最小：



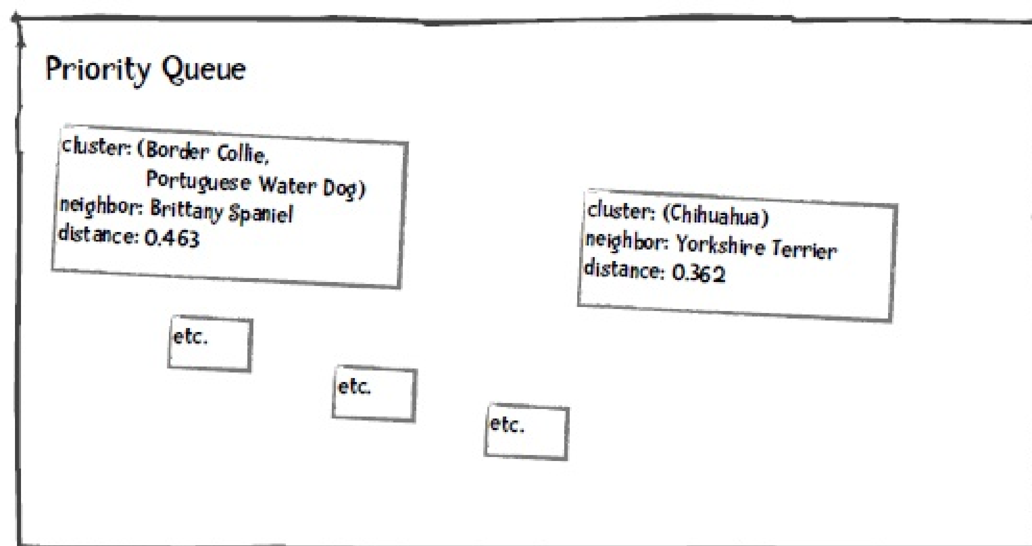
我们看看Python中如何使用优先队列：

```
>>> from Queue import PriorityQueue                  # 加载优先队列类
>>> singersQueue = PriorityQueue()                   # 创建对象
>>> singersQueue.put((16, 'Suzuka Nakamoto')) # 插入元素
>>> singersQueue.put((15, 'Moa Kikuchi'))
>>> singersQueue.put((14, 'Yui Mizuno'))
>>> singersQueue.put((17, 'Ayaka Sasaki'))
>>> singersQueue.get() # 获取第一个元素，即最年轻的歌手Yui。
(14, 'Yui Mizuno')
>>> singersQueue.get()
(15, 'Moa Kikuchi')
>>> singersQueue.get()
(16, 'Suzuka Nakamoto')
>>> singersQueue.get()
(17, 'Ayaka Sasaki')
```

在进行聚类时，我们将分类、离它最近的分类、以及距离插入到优先队列中，距离作为优先级。比如上面的犬种示例，Border Collie最近的分类是Portuguese WD，距离是0.232：



我们将优先队列中距离最小的两个分类取出来，合并成一个分类，并重新插入到优先队列中。比如下图是将Border Collie和Portuguese WD合并后的结果：



重复这个过程，直到队列中只有一个元素为止。当然，我们插入的数据会复杂一些，请看下面的讲解。

从文件中读取数据

数据文件是CSV格式的（以逗号分隔），第一行是列名，第一列是犬种，第二列之后是特征值：

```
breed,height (inches),weight (pounds)
Border Collie,20,45
Boston Terrier,16,20
Brittany Spaniel,18,35
Bullmastiff,27,120
Chihuahua,8,8
German Shepherd,25,78
Golden Retriever,23,70
Great Dane,32,160
Portuguese Water Dog,21,50
Standard Poodle,19,65
Yorkshire Terrier,6,7
```

我们用Python的列表结构来存储这些数据，data[0]用来存放所有记录的分类，如data[0][0]是Border Collie，data[0][1]是Boston Terrier。data[1]则是所有记录的高度，data[2]是重量。特征列的数据都会转换成浮点类型，如data[1][0]是20.0，data[2][0]是45.0等。在读取数据时就需要对其进行标准化。此外，我们接下来会使用“下标”这个术语，如第一条记录Border Collie的下标是0，第二条记录Boston Terrier下标是1等。

初始化优先队列

以Border Collie为例，我们需要计算它和其它犬种的距离，保存在Python字典里：

```
{1: ((0, 1), 1.0244),  # Border Collie（下标为0）和Boston Terrier（下标为1）之间的距离为1.0244
 2: ((0, 2), 0.463),   # Border Collie和Brittany Spaniel（下标为2）之间的距离为0.463
 ...
 10: ((0, 10), 2.756)} # Border Collie和Yorkshire Terrier的距离为2.756
```

此外，我们会记录Border Collie最近的分类及距离：这对犬种是(0, 8)，即下标为0的Border Collie和下标为8的Portuguese WD，距离是0.232。

距离相等的问题以及为何要使用元组

你也许注意到了，Portuguese WD和Standard Poodle的距离是0.566，Boston Terrier和Brittany Spaniel的距离也是0.566，如果我们通过最短距离来取，很可能会取出Standard Poodle和Boston Terrier进行组合，这显然是错误的，所以我们才会使用元组来存放这对犬种的下标，以作判断。比如说，Portuguese WD的记录是：

```
['Portuguese Water Dog', 0.566, (8, 9)]
```

它的近邻Standard Poodle的记录是：

```
['Standard Poodle', 0.566, (8, 9)]
```

我们可以通过这个元组来判断这两条记录是否是一对。

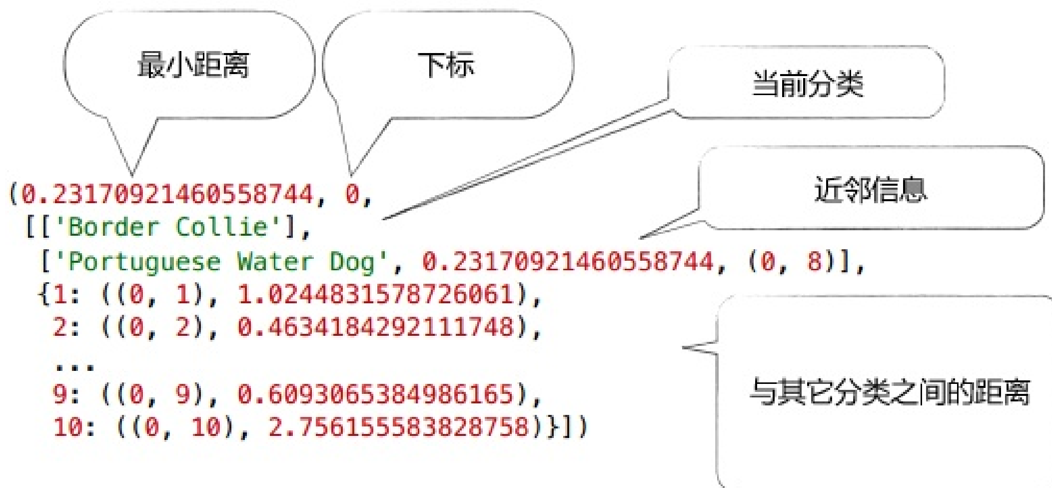
距离相等的另一个问题

在介绍优先队列时，我用了歌手的年龄举例，如果他们的年龄相等，取出的顺序又是怎样的呢？

```
>>> singersQueue.put((15,'Suzuka Nakamoto'))
>>> singersQueue.put((15,'Moa Kikuchi'))
>>> singersQueue.put((15, 'Yui Mizuno'))
>>> singersQueue.put((15, 'Avaka Sasaki'))
>>> singersQueue.put((12, 'Megumi Okada'))
>>> singersQueue.get()
(12, 'Megumi Okada')
>>> singersQueue.get()
(15, 'Avaka Sasaki')
>>> singersQueue.get()
(15, 'Moa Kikuchi')
>>> singersQueue.get()
(15, 'Suzuka Nakamoto')
>>> singersQueue.get()
(15, 'Yui Mizuno')
>>>
```

可以看到，如果年龄相等，优先队列会根据记录中的第二个元素进行判断，即歌手的姓名，并按字母顺序返回，如Avaka会比Moa优先返回。

在犬种示例中，我们让距离成为第一优先级，下标成为第二优先级。因此，我们插入到优先队列的一条完整记录是这样的：



重复下述步骤，直到仅剩一个分类

我们从优先队列中取出两个元素，对它们进行合并。如合并Border Collie和Portuguese WD后，会形成一个新的分类：

```
['Border Collie', 'Portuguese WD']
```

然后我们需要计算新的分类和其它分类之间的距离，方法是对取出的两个分类的距离字典进行合并。如第一个分类的距离字段是distanceDict1，第二个分类的是distanceDict2，新的距离字段是newDistanceDict：

```
初始化newDistanceDict
对于distanceDict1的每一个键值对:
    如果这个键在distanceDict2中存在:
        如果这个键在distanceDict1中的距离要比在distanceDict2中的距离小:
            将distanceDict1中的距离存入newDistanceDict
        否则:
            将distanceDict2中的距离存入newDistanceDict
```

key	value in the Border Collie Distance List	value in the Portuguese Water Dog Distance List	value in the Distance List for the new cluster
0	–	((0, 8), 0.2317092146055)	–
1	((0, 1), 1.02448315787260)	((1, 8), 1.25503395239308)	((0, 1), 1.02448315787260)
2	((0, 2), 0.46341842921117)	((2, 8), 0.69512764381676)	(0, 2), 0.46341842921117)
3	((0, 3), 2.52128307411504)	((3, 8), 2.3065500082408)	((3, 8), 2.3065500082408)
4	((0, 4), 2.41700998092941)	((4, 8), 2.643745991701)	((0, 4), 2.41700998092941)
5	((0, 5), 1.31725590972761)	((5, 8), 1.088215707936)	((5, 8), 1.088215707936)
6	((0, 6), 0.90660838225252)	((6, 8), 0.684696194462)	((6, 8), 0.684696194462)
7	((0, 7), 3.98523295438990)	((7, 8), 3.765829069545)	((7, 8), 3.765829069545)
8	((0, 8), 0.23170921460558)	–	–
9	((0, 9), 0.60930653849861)	((8, 9), 0.566225873458)	((8, 9), 0.566225873458)
10	((0, 10), 2.7561555838287)	((8, 10), 2.980333906137)	((0, 10), 2.7561555838287)

经过计算后，插入到优先队列中的新分类的完整记录是：

```
(0.4634184292111748, 11, [('Border Collie', 'Portuguese Water Dog'),
 [2, 0.4634184292111748, (0, 2)],
 {1: ((0, 1), 1.0244831578726061), 2: ((0, 2), 0.4634184292111748),
  3: ((3, 8), 2.306550008240866), 4: ((0, 4), 2.4170099809294157),
  5: ((5, 8), 1.0882157079364436), 6: ((6, 8), 0.6846961944627522),
  7: ((7, 8), 3.7658290695451373), 9: ((8, 9), 0.5662258734585477),
  10: ((0, 10), 2.756155583828758)}])
```

代码实践

你能将上面的算法用Python实现吗？你可以从hierarchicalClustererTemplate.py这个文件开始，完成以下步骤：

1. 编写init方法，对于每条记录：
    i. 计算该分类和其它分类之间的欧几里得距离；
    ii. 找出该分类的近邻；
    iii. 将这些信息放到优先队列的中。

2. 编写cluster方法，重复以下步骤，直至剩下一个分类：
   i. 从优先队列中获取两个元素；
   ii. 合并；
   iii. 将合并后的分类放回优先队列中。



解答

> 注意，我的实现并不一定是最好的，你可以写出更好的！

```
from queue import PriorityQueue
import math

"""
层次聚类示例代码
"""

def getMedian(alist):
    """计算中位数"""
    tmp = list(alist)
    tmp.sort()
    alen = len(tmp)
    if (alen % 2) == 1:
        return tmp[alen // 2]
    else:
        return (tmp[alen // 2] + tmp[(alen // 2) - 1]) / 2

def normalizeColumn(column):
    """计算修正的标准分"""
    median = getMedian(column)
    asd = sum([abs(x - median) for x in column]) / len(column)
    result = [(x - median) / asd for x in column]
    return result

class hClusterer:
    """该聚类器默认数据的第一列是标签，其它列是数值型的特征。"""

    def __init__(self, filename):
        file = open(filename)
        self.data = {}
        self.counter = 0
        self.queue = PriorityQueue()
        lines = file.readlines()
        file.close()
        header = lines[0].split(',')
        self.cols = len(header)
        self.data = [[] for i in range(len(header))]
```

```
        for line in lines[1:]:
            cells = line.split(',')
            toggle = 0
            for cell in range(self.cols):
                if toggle == 0:
                   self.data[cell].append(cells[cell])
                   toggle = 1
                else:
                    self.data[cell].append(float(cells[cell]))
        # 标准化特征列（即跳过第一列）
        for i in range(1, self.cols):
                self.data[i] = normalizeColumn(self.data[i])

        ###
        ###  数据已经读入内存并做了标准化，对于每一条记录，将执行以下步骤：
        ###     1. 计算该分类和其他分类的距离，如当前分类的下标是1，
        ###        它和下标为2及下标为3的分类之间的距离用以下形式表示：
        ###        {2: ((1, 2), 1.23),  3: ((1, 3), 2.3)... }
        ###     2. 找出距离最近的分类；
        ###     3. 将该分类插入到优先队列中。
        ###

        # 插入队列
        rows = len(self.data[0])

        for i in range(rows):
            minDistance = 99999
            nearestNeighbor = 0
            neighbors = {}
            for j in range(rows):
                if i != j:
                    dist = self.distance(i, j)
                    if i < j:
                        pair = (i,j)
                    else:
                        pair = (j,i)
                    neighbors[j] = (pair, dist)
                    if dist < minDistance:
                        minDistance = dist
                        nearestNeighbor = j
                        nearestNum = j
            # 记录这两个分类的配对信息
            if i < nearestNeighbor:
                nearestPair = (i, nearestNeighbor)
            else:
                nearestPair = (nearestNeighbor, i)

            # 插入优先队列
            self.queue.put((minDistance, self.counter,
                            [[self.data[0][i]], nearestPair, neighbors]))
            self.counter += 1

    def distance(self, i, j):
        sumSquares = 0
        for k in range(1, self.cols):
            sumSquares += (self.data[k][i] - self.data[k][j])**2
        return math.sqrt(sumSquares)

    def cluster(self):
         done = False
         while not done:
             topOne = self.queue.get()
             nearestPair = topOne[2][1]
             if not self.queue.empty():
                 nextOne = self.queue.get()
                 nearPair = nextOne[2][1]
                 tmp = []

                 ##  我从队列中取出了两个元素：topOne和nextOne，
                 ##  检查这两个分类是否是一对，如果不是就继续从优先队列中取出元素，
                 ##  直至找到topOne的配对分类为止。
                 while nearPair != nearestPair:
```

```
                    tmp.append((nextOne[0], self.counter, nextOne[2]))
                    self.counter += 1
                    nextOne = self.queue.get()
                    nearPair = nextOne[2][1]

                ## 将不处理的元素退回给优先队列
                for item in tmp:
                    self.queue.put(item)

                if len(topOne[2][0]) == 1:
                    item1 = topOne[2][0][0]
                else:
                    item1 = topOne[2][0]
                if len(nextOne[2][0]) == 1:
                    item2 = nextOne[2][0][0]
                else:
                    item2 = nextOne[2][0]
                ##  curCluster即合并后的分类
                curCluster = (item1, item2)

                ## 对于这个新的分类需要做两件事情：首先找到离它最近的分类，然后合并距离字典。
                ## 如果item1和元素23的距离是2，item2和元素23的距离是4，我们取较小的那个距离，即单链
                minDistance = 99999
                nearestPair = ()
                nearestNeighbor = ''
                merged = {}
                nNeighbors = nextOne[2][2]
                for (key, value) in topOne[2][2].items():
                   if key in nNeighbors:
                       if nNeighbors[key][1] < value[1]:
                            dist =  nNeighbors[key]
                       else:
                           dist = value
                       if dist[1] < minDistance:
                            minDistance =  dist[1]
                            nearestPair = dist[0]
                            nearestNeighbor = key
                       merged[key] = dist

                if merged == {}:
                   return curCluster
                else:
                   self.queue.put( (minDistance, self.counter,
                                    [curCluster, nearestPair, merged]))
                   self.counter += 1

def printDendrogram(T, sep=3):
    """打印二叉树状图。树的每个节点是一个二元组。这个方法摘自：
    http://code.activestate.com/recipes/139422-dendrogram-drawing/"""

    def isPair(T):
        return type(T) == tuple and len(T) == 2

    def maxHeight(T):
        if isPair(T):
            h = max(maxHeight(T[0]), maxHeight(T[1]))
        else:
            h = len(str(T))
        return h + sep

    activeLevels = {}

    def traverse(T, h, isFirst):
        if isPair(T):
            traverse(T[0], h-sep, 1)
            s = [' ']*(h-sep)
            s.append('|')
        else:
            s = list(str(T))
            s.append(' ')

        while len(s) < h:
```

```
            s.append('-')

        if (isFirst >= 0):
            s.append('+')
            if isFirst:
                activeLevels[h] = 1
            else:
                del activeLevels[h]

        A = list(activeLevels)
        A.sort()
        for L in A:
            if len(s) < L:
                while len(s) < L:
                    s.append(' ')
                s.append('|')

        print (''.join(s))

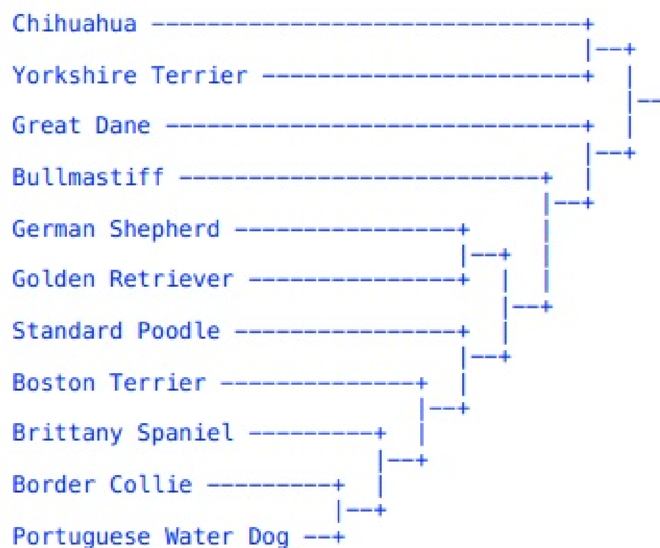
        if isPair(T):
            traverse(T[1], h-sep, 0)

    traverse(T, maxHeight(T), -1)


filename = '/Users/raz/Dropbox/guide/data/dogs.csv'

hg = hClusterer(filename)
cluster = hg.cluster()
printDendrogram(cluster)
```

运行结果和我们手算的一致：



动手实践

这里提供了77种早餐麦片的营养信息，包括以下几项：

- 麦片名称
- 热量
- 蛋白质
- 脂肪

- 纳
- 纤维
- 碳水化合物
- 糖
- 钾
- 维生素



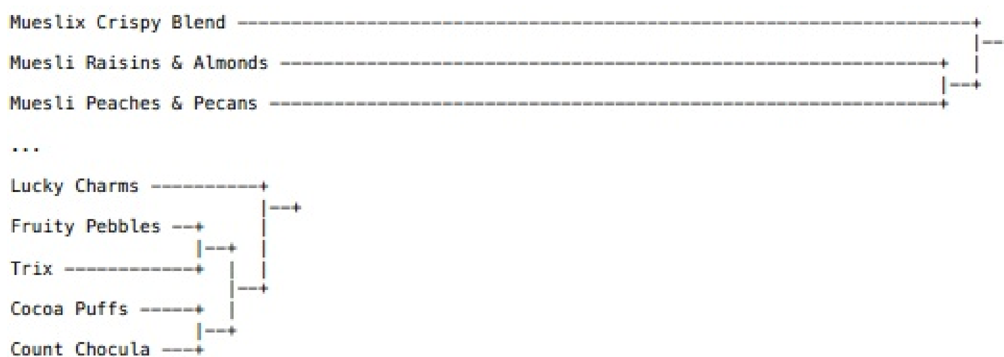
请对这个数据集进行层次聚类：

- 哪种麦片和Trix最相近？
- 与Muesli Raisins & Almonds最相近的是？

> 数据集来自：http://lib.stat.cmu.edu/DASL/Datafiles/Cereals.html

结果

我们只需将代码中的文件名替换掉就可以了，结果如下：



```
Mueslix Crispy Blend ---------------------------------------------------------+
                                                                               |--+
Muesli Raisins & Almonds -----------------------------------------------------+ |
                                                                            |--+
Muesli Peaches & Pecans -----------------------------------------------------+

...

Lucky Charms ----------+
                       |--+
Fruity Pebbles --+     |
                 |--+  |
Trix ------------+  |  |
                    |--+
Cocoa Puffs -----+  |
                 |--+
Count Chocula ---+
```

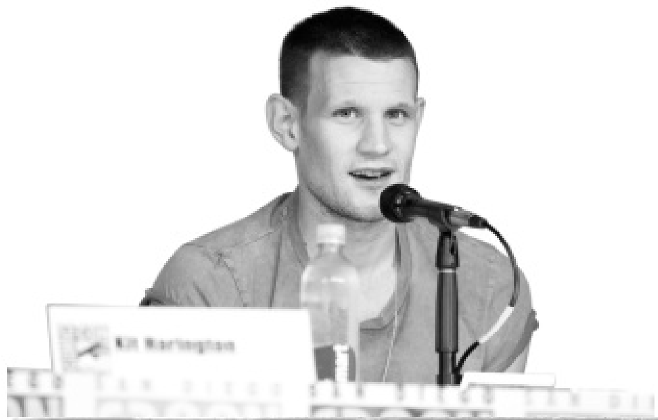
因此Trix和Fruity Pebbles最相似（你可以去买这两种麦片尝尝）。Muesli Raisins & Almonds和Muesli Peaches & Pecans最相似。



> 好了，这就是层次聚类算法，很简单吧！

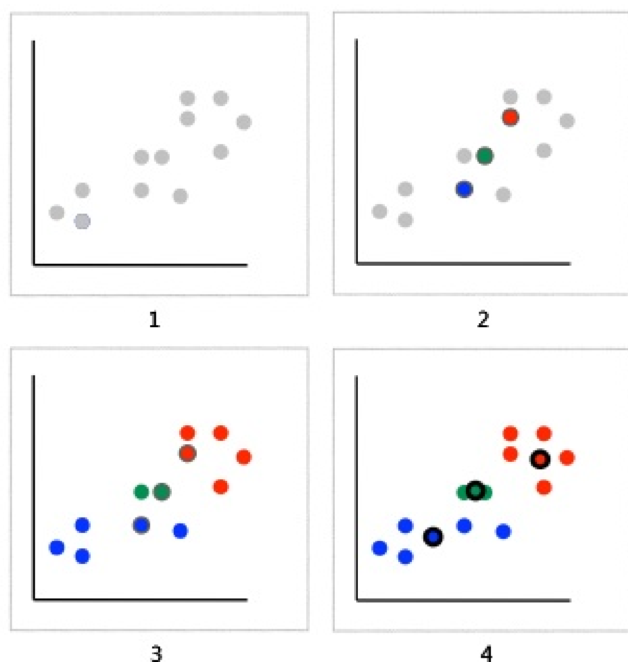
k-means聚类算法

使用k-means算法时需要指定分类的数量，这也是算法名称中“k”的由来。



> k-means是Lloyd博士在1957年提出的，虽然这个算法已有50年的历史，但却是当前最流行的聚类算法！

下面让我们来了解一下k-means聚类过程：



1. 我们想将图中的记录分成三个分类（即k=3），比如上文提到的犬种数据，坐标轴分别是身高和体重。
2. 由于k=3，我们随机选取三个点来作为聚类的起始点（分类的中心点），并用红黄蓝三种颜色标识。
3. 然后，我们根据其它点到中心点的距离来进行分配，这样就能将这些点分成三类了。
4. 计算这些分类的中心点，以此作为下一次计算的起始点。重复这个过程，直到中心点不再变动，或迭代次数超过某个阈值为止。

所以k-means算法可概括为：

1. 随机选取k个元素作为中心点；
2. 根据距离将各个点分配给中心点；
3. 计算新的中心点；
4. 重复2、3，直至满足条件。

我们来看一个示例，将以下点分成两个分类：

```
(1, 2)
(1, 4)
(2, 2)
(2, 3)
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

第一步 随机选取中心点

我们选取(1, 4)作为分类1的中心点，(4, 2)作为分类2的中心点；

第二步 将各点分配给中心点

可以用各类距离计算公式，为简单起见，这里我们使用曼哈顿距离：

point	distance from centroid 1 (1, 4)	distance from centroid 2 (4, 2)
(1, 2)	2	3
(1,4)	0	5
(2, 2)	3	2
(2, 3)	2	3
(4, 2)	5	0
(4, 4)	3	2
(5, 1)	7	2
(5, 3)	5	2

聚类结果如下：

```
CLUSTER 1
(1, 2)
(1, 4)
(2, 3)
```

```
CLUSTER 2
(2, 2)
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

第三步 更新中心点

通过计算平均值来更新中心点，如x轴的均值是：

(1 + 1 + 2) / 3 = 4 / 3 = 1.33

y轴是：

(2 + 4 + 3) / 3 = 9 / 3 = 3

因此分类1的中心点是(1.33, 3)。计算得到分类2的中心点是(4, 2.4)。

第四步 重复前面两步

两个分类的中心点由(1, 4)、(4, 2)变为了(1.33, 3)、(4, 2.4)，我们使用新的中心点重新计算。

重复第二步 将各点分配给中心点

同样是计算曼哈顿距离：

point	distance from centroid 1 (1.33, 3)	distance from centroid 2 (4, 2.4)
(1, 2)	1.33	3.4
(1, 4)	1.33	4.6
(2, 2)	1.67	2.4
(2, 3)	0.67	2.6
(4, 2)	3.67	0.4
(4, 4)	3.67	1.6
(5, 1)	5.67	2.4
(5, 3)	3.67	1.6

新的聚类结果是：

```
CLUSTER 1
(1, 2)
(1, 4)
(2, 2)
(2, 3)
```

```
CLUSTER 2
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

重复第三步 更新中心点

- 分类1：(1.5, 2.75)
- 分类2：(4.5, 2.5)

重复第二步 将各点分配给中心点

point	distance from centroid 1 (1.5, 2.75)	distance from centroid 2 (4.5, 2.5)
(1, 2)	1.25	4.0
(1, 4)	1.75	5.0
(2, 2)	1.25	3.0
(2, 3)	0.75	3.0
(4, 2)	3.25	1.0
(4, 4)	3.75	2.0
(5, 1)	5.25	2.0
(5, 3)	3.75	1.0

```
CLUSTER 1
(1, 2)
(1, 4)
(2, 2)
(2, 3)
```

```
CLUSTER 2
(4, 2)
(4, 4)
(5, 1)
(5, 3)
```

重复第三步 更新中心点

- 分类1：(1.5, 2.75)
- 分类2：(4.5, 2.5)

可以看到中心点并没有改变，所以计算也就结束了。



> 当中心点不再变化时，或者说不再有某个点从一个分类转移到另一个分类时，我们就会停止计算。这个时候我们称该算法已经收敛。算法运行过程中，中心点的大幅转移是在前几次迭代中产生的，后面的迭代中变动的幅度就会减小。也就是说，k-means算法的重点是在前期迭代，而后期的迭代只是细微的调整。



> 基于k-means的这种特点，我们可以将“没有点发生转移”弱化成“少于1%的点发生转移”来作为计算停止条件，这也是最普遍的做法。



> k-means好简单呀！

扩展阅读

k-means是一种最大期望算法，这类算法会在“期望”和“最大化”两个阶段不断迭代。比如k-means的期望阶段是将各个点分配到它们所“期望”的分类中，然后在最大化阶段重新计算中心点的位置。如果你对此感兴趣，可以前去阅读维基百科上的词条。

登山式算法

再继续讨论k-means算法之前，我想先介绍一下登山式算法。



假设我们想要登上一座山的顶峰，可以通过以下步骤实现：

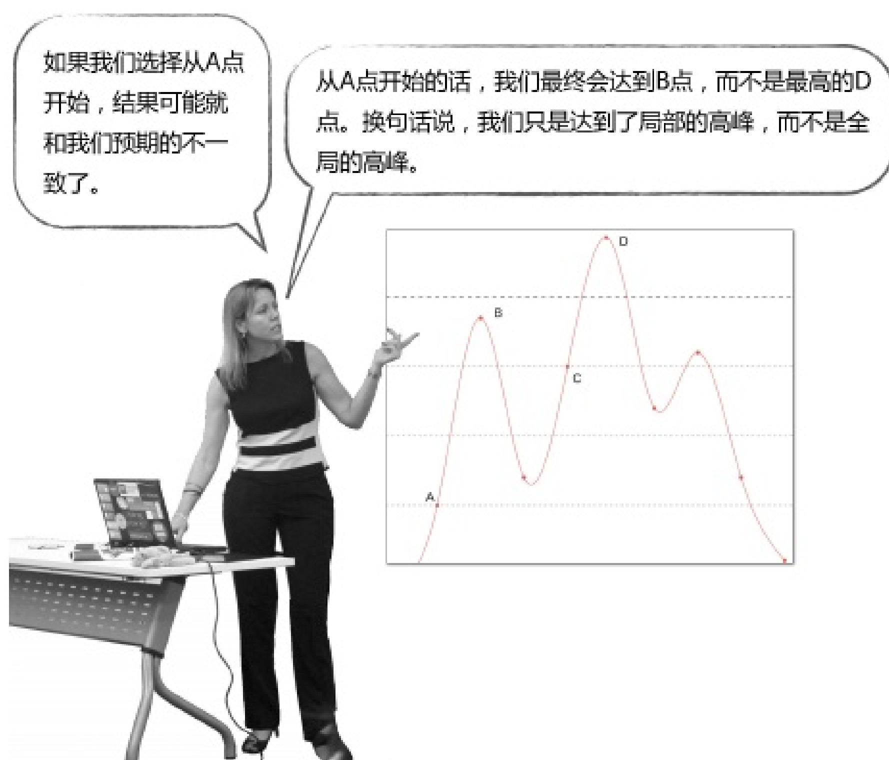
1. 在山上随机选取一个点作为开始；
2. 向高处爬一点；
3. 重复第2步，直到没有更高的点。

这种做法看起来很合理，比如对于下图所示的山峰：



无论我们从哪个点开始攀登，最终都可以达到顶峰。

但对于下面这张图：



所以说，这种简单的登山式算法并不一定能得到最优解。

k-means就是这样一种算法，它不能保证最终结果是最优的，因为我们一开始选择的中心点是随机的，很有可能就会选到上面的A点，最终获得局部最优解B点。因此，最终的聚类结果和起始点的选择有很大关系。但尽管如此，k-means通常还是能够获得良好的结果的。



> 那我们如何比较不同的聚类结果呢？

误差平方和（SSE）

我们可以使用误差平方和（或称离散程度）来评判聚类结果的好坏，它的计算方法是：计算每个点到中心点的距离平方和。

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} dist(c_i, x)^2$$

上面的公式中，第一个求和符号是遍历所有的分类，比如i=1时计算第一个分类，i=2时计算第二个分类，直到计算第k个分类；第二个求和符号是遍历分类中所有的点；Dist指代距离计算公式（如曼哈顿距离、欧几里得距离）；计算数据点x和中心点c_i之间的距离，平方后相加。

假设我们对同一数据集使用了两次k-means聚类，每次选取的起始点不一样，想知道最后得到的聚类结果哪个更优，就可以计算和比较SSE，结果小的效果好。



> 下面让我们开始编程吧！

```
import math
import random


"""
K-means算法
"""

def getMedian(alist):
    """计算中位数"""
    tmp = list(alist)
    tmp.sort()
    alen = len(tmp)
    if (alen % 2) == 1:
        return tmp[alen // 2]
    else:
        return (tmp[alen // 2] + tmp[(alen // 2) - 1]) / 2


def normalizeColumn(column):
    """计算修正的标准分"""
    median = getMedian(column)
    asd = sum([abs(x - median) for x in column]) / len(column)
    result = [(x - median) / asd for x in column]
    return result


class kClusterer:
    """kMeans聚类算法，第一列是分类，其余列是数值型特征"""

    def __init__(self, filename, k):
        """k是分类的数量，该函数完成以下功能：
           1. 读取filename的文件内容
           2. 按列存储到self.data变量中
           3. 计算修正的标准分
           4. 随机选取起始点
           5. 将各个点分配给中心点
        """
        file = open(filename)
        self.data = {}
        self.k = k
        self.counter = 0
        self.iterationNumber = 0
        # 用于跟踪本次迭代有多少点的分类发生了变动
```

```
        self.pointsChanged = 0
        # 误差平方和
        self.sse = 0
        #
        # 读取文件
        #
        lines = file.readlines()
        file.close()
        header = lines[0].split(',')
        self.cols = len(header)
        self.data = [[] for i in range(len(header))]
        # 按列存储数据，如self.data[0]是第一列的数据，
        # self.data[0][10]是第一列第十行的数据。
        for line in lines[1:]:
            cells = line.split(',')
            toggle = 0
            for cell in range(self.cols):
                if toggle == 0:
                   self.data[cell].append(cells[cell])
                   toggle = 1
                else:
                    self.data[cell].append(float(cells[cell]))

        self.datasize = len(self.data[1])
        self.memberOf = [-1 for x in range(len(self.data[1]))]
        #
        # 标准化
        #
        for i in range(1, self.cols):
                self.data[i] = normalizeColumn(self.data[i])

        # 随机选取起始点
        random.seed()
        self.centroids = [[self.data[i][r]  for i in range(1, len(self.data))]
                           for r in random.sample(range(len(self.data[0])),
                                                 self.k)]
        self.assignPointsToCluster()


    def updateCentroids(self):
        """根据分配结果重新确定聚类中心点"""
        members = [self.memberOf.count(i) for i in range(len(self.centroids))]
        self.centroids = [[sum([self.data[k][i]
                                for i in range(len(self.data[0]))
                                if self.memberOf[i] == centroid])/members[centroid]
                           for k in range(1, len(self.data))]
                          for centroid in range(len(self.centroids))]



    def assignPointToCluster(self, i):
        """根据距离计算所属中心点"""
        min = 999999
        clusterNum = -1
        for centroid in range(self.k):
            dist = self.euclideanDistance(i, centroid)
            if dist < min:
                min = dist
                clusterNum = centroid
        # 跟踪变动的点
        if clusterNum != self.memberOf[i]:
            self.pointsChanged += 1
        # 计算距离平方和
        self.sse += min**2
        return clusterNum

    def assignPointsToCluster(self):
        """分配所有的点"""
        self.pointsChanged = 0
        self.sse = 0
        self.memberOf = [self.assignPointToCluster(i)
```

```
                          for i in range(len(self.data[1]))]


    def euclideanDistance(self, i, j):
        """计算欧几里得距离"""
        sumSquares = 0
        for k in range(1, self.cols):
            sumSquares += (self.data[k][i] - self.centroids[j][k-1])**2
        return math.sqrt(sumSquares)

    def kCluster(self):
        """开始进行聚类，重复以下步骤：
        1. 更新中心点
        2. 重新分配
        直至变动的点少于1%。
        """
        done = False

        while not done:
            self.iterationNumber += 1
            self.updateCentroids()
            self.assignPointsToCluster()
            #
            # 如果变动的点少于1%则停止迭代
            #
            if float(self.pointsChanged) / len(self.memberOf) <  0.01:
                done = True
        print("Final SSE: %f" % self.sse)

    def showMembers(self):
        """输出结果"""
        for centroid in range(len(self.centroids)):
             print ("\n\nClass %i\n========" % centroid)
             for name in [self.data[0][i]  for i in range(len(self.data[0]))
                          if self.memberOf[i] == centroid]:
                 print (name)

##
## 对犬种数据进行聚类，令k=3
###
# 请自行修改文件路径
km = kClusterer('../../data/dogs.csv', 3)
km.kCluster()
km.showMembers()
```



> 我们来分析一下这段代码。

犬种数据用表格来展示是这样的，身高和体重都做了标准化：

breed	height	weight
Border Collie	0	-0.1455
Boston Terrier	-0.7213	-0.873
Brittany Spaniel	-0.3607	-0.4365
Bullmastiff	1.2623	2.03704
German Shepherd	0.9016	0.81481
...	...	...

因为需要按列存储，转化后的Python格式是这样的：

```
data = [['Border Collie', 'Bosten Terrier', 'Brittany Spaniel', ...],
        [0, -0.7213, -0.3607, ...],
        [-0.1455, -0.7213, -0.4365, ...],
        ...]
```

我们在层次聚类中用的也是此法，这样做的好处是能够方便地应用不同的数学函数。比如计算中位数和计算标准分的函数，都是以列表作为参数的：

```
>>> normalizeColumn([8, 6, 4, 2])
[1.5, 0.5, -0.5, -1.5]
```

`__init__` 函数首先将文件读入进来，按列存储，并进行标准化。随后，它会选取k个起始点，并将记录中的点分配给这些中心点。`kCluster` 函数则开始迭代计算中心点的新位置，直到少于1%的点发生变动为止。

程序的运行结果如下：

```
Final SSE: 5.243159

Class 0
=======
Bullmastiff
Great Dane

Class 1
=======
Boston Terrier
Chihuahua
Yorkshire Terrier

Class 2
=======
Border Collie
Brittany Spaniel
German Shepherd
Golden Retriever
Portuguese Water Dog
Standard Poodle
```

聚类结果非常棒！

动手实践

用上面的聚类程序来对麦片数据集进行聚类，令k=4，并回答以下问题：

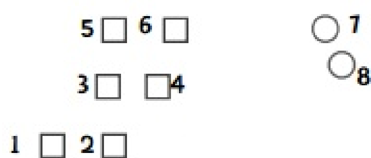
1. 甜味麦片都被聚类到一起了吗，如Cap'n'Crunch, Cocoa Puffs, Froot Loops, Lucky Charms？
2. 麸类麦片聚到一起了吗，如100% Bran, All-Bran, All-Bran with Extra Fiber, Bran Chex？
3. Cheerios被分到了哪个类别，是不是一直和Special K一起？

再来对加仑公里数的数据进行聚类，令k=8。运行结果大致令人满意，但有时候会出现记录数为空的分类。

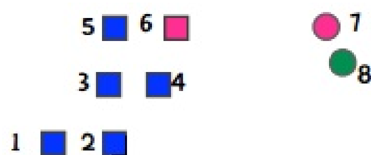


> 我要求聚类成8个分类，但其中一个是空的，肯定代码有问题！

我们用示例来看这个问题，假设需要将以下8个点分成3个类别：



我们选取1、7、8作为起始点，因此第一次聚类的结果是：



随后，我们重新计算中心点，即下图中的加号：



这时，6离蓝色中心点较近，7离绿色中心点较近，因此粉色的分类就为空了。

所以说，虽然我们指定了k的值，但不代表最终结果就会有k个分类。这通常是好事，比如上面的例子中，看起来就应该要分成两类。如果有1000条数据，我们指定k=10，但结果有两个为空，那很有可能这个数据集本来就该分成8个类别，因此可以尝试用k=8来重新计算。

另一方面，如果你要求分类都不为空，那就需要改变一下算法：当发现空的分类时，就重新指定这个分类的中心点。一种做法是选取离这个中心点最远的点，比如上面的例子中，发现粉色分类为空，就将中心点变为点1，因为它离粉色中心点最远。





k-means++

前面我们提到k-means是50年代发明的算法，它的实现并不复杂，但仍是现今最流行的聚类算法。不过它也有一个明显的缺点。在算法一开始需要随机选取k个起始点，正是这个随机会有问题。有时选取的点能产生最佳结果，而有时会让结果变得很差。k-means++则改进了起始点的选取过程，其余的和k-means一致。

以下是k-means++选取起始点的过程：

1. 随机选取一个点；
2. 重复以下步骤，直到选完k个点：
    i. 计算每个数据点（dp）到各个中心点的距离（D），选取最小的值，记为D(dp)；
    ii. 根据D(dp)的概率来随机选取一个点作为中心点。

我们来讲解一下何为“根据D(dp)的概率来随机选取”。假设选取过程进行到一半，已经选出了两个点，现在需要选第三个。假设还有五个点可供选择，它们离已有的两个中心点的距离是：

	Dc1	Dc2
dp1	5	7
dp2	9	8
dp3	2	5
dp4	3	7
dp5	5	2

*Dc1*表示到中心点*1*的距离，*Dc2*表示到中心点*2*的距离。

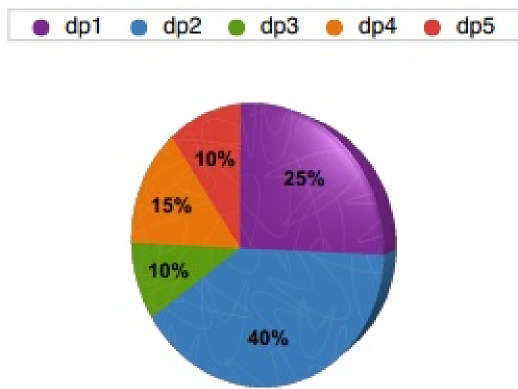
我们选取最小的距离：

	closest
dp1	5
dp2	8
dp3	2
dp4	3
dp5	2

然后将这些数值转化成总和为1的权重值，做法就是将每个距离除以距离的和（20），得到：

	weight
dp1	0.25
dp2	0.40
dp3	0.10
dp4	0.15
dp5	0.10
sum	1.00

我们可以通过转盘游戏来理解：



比如我们扔个球到这个转盘里，它停在哪个颜色就选取这个点作为新的中心点。这就叫做“根据D(dp)的概率来随机选取”。

比如我们有以下Python数据：

```
data = [('dp1', 0.25), ('dp2', 0.4), ('dp3', 0.1),
        ('dp4', 0.15), ('dp5', 0.1)]
```

然后来编写一个函数来完成选取过程：

```
import random
random.seed()

def roulette(datalist):
    i = 0
    soFar = datalist[0][1]
    ball = random.random()
    while soFar < ball:
        i += 1
        soFar += datalist[i][1]
    return datalist[i][0]
```

如果这个函数运行正确，我们选取100次的话，其中25次应该是dp1，40次是dp2，10次是dp3，15次是dp4，10次是dp5。让我们来测试一下：

```
import collections
results = collections.defaultdict(int)
for i in range(100):
    results[roulette(data)] += 1
print results

{'dp5': 11, 'dp4': 15, 'dp3': 10, 'dp2': 38, 'dp1': 26}
```

结果是符合预期的！

k-means++选取起始点的方法总结下来就是：第一个点还是随机的，但后续的点就会尽量选择离现有中心点更远的点。



> 好了，下面让我们开始写代码吧！

代码实践

你能用Python实现k-means++算法吗？k-means++和k-means的唯一区别就是起始点的选取过程，你需要做的是将下面的代码：

```
self.centroids = [[self.data[i][r] for i in range(1, len(self.data))]
                   for r in random.sample(range(len(self.data[0])),
                                              self.k)]
```

替换为：

```
self.selectInitialCentroids()
```

你的任务就是编写这个函数！



解答

```
def distanceToClosestCentroid(self, point, centroidList):
    result = self.eDistance(point, centroidList[0])
    for centroid in centroidList[1:]:
        distance = self.eDistance(point, centroid)
        if distance < result:
            result = distance
    return result

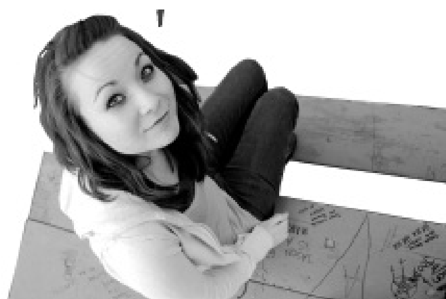
def selectInitialCentroids(self):
    """实现k-means++算法中的起始点选取过程"""
    centroids = []
    total = 0
    # 首先随机选取一个点
    current = random.choice(range(len(self.data[0])))
    centroids.append(current)
    # 开始选取剩余的点
    for i in range(0, self.k - 1):
        # 计算每个点到最近的中心点的距离
        weights = [self.distanceToClosestCentroid(x, centroids)
                   for x in range(len(self.data[0]))]
        total = sum(weights)
        # 转换为权重
        weights = [x / total for x in weights]
        # 开始随机选取
        num = random.random()
        total = 0
        x = -1
        # 模拟轮盘游戏
        while total < num:
            x += 1
            total += weights[x]
        entroids.append(x)
    self.centroids = [[self.data[i][r]  for i in range(1, len(self.data))]
                      for r in centroids]
```

安然事件

你应该还对这件事有些印象吧？安然公司曾是一家超大型企业，有着千亿元的收入和两万名员工（微软只有220亿收入）。由于管理体制的破败和受贿，包括人为制造能源危机致使加州大停电，安然公司最终面临破产，大批人员被判入狱。有一部名为“The Smartest Guys in the Room”的纪录片，读者可以到Netflix或亚马逊上观看。



> 安然事件的确挺有趣的，不过这和数据挖掘有什么关系呢？



> 在调查过程中，美国联邦能源管理委员会收获了60万封公司内部邮件。这些邮件可以从网络上下载：
>
> http://en.wikipedia.org/wiki/Enron_Corpus
>
> https://www.cs.cmu.edu/~./enron/

我们来用其中的一小部分数据来举例，下表整理了一些公司人员互通邮件的次数：

	Kay	Chris	Sara	Tana	Steven	Mark
Kay	0	53	37	6	0	12
Chris	53	0	1	0	2	0
Sara	37	1	0	1144	0	962
Tana	6	0	1144	0	0	1201
Steven	0	0	2	0	0	0
Mark	12	0	962	1201	0	0

可以在这里下载缩减后的数据集。完整的数据在这里，超过300MB。

动手实践

你能使用层次聚类算法将这些人分成若干类别吗？

解答

我们会通过两个人收发邮件的对象来计算相似度。比如我经常和Ann、Ben、Clara等人通信，你也一样，那么我俩就是相似的：

between ->	Ann	Ben	Clara	Dongmei	Emily	Frank
my emails	127	25	119	5	1	6
your emails	172	35	123	7	3	5

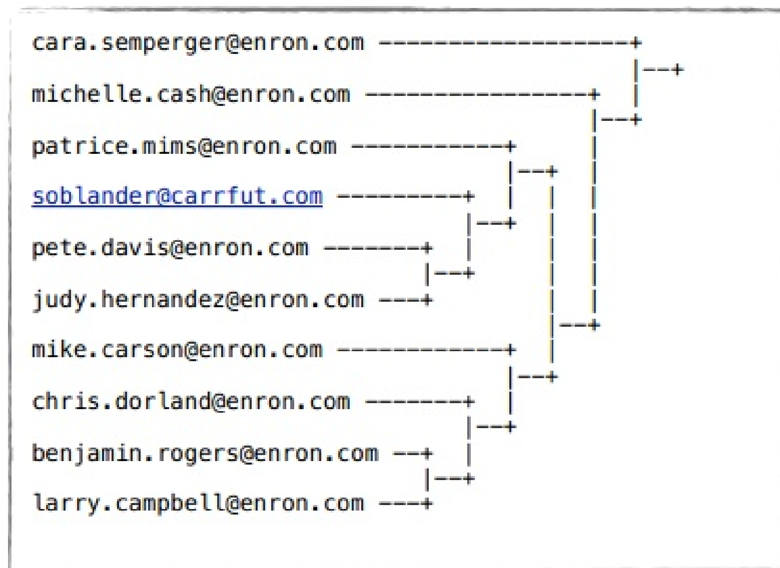
但如果将你我之间的通信也计算进去：

between ->	me	you	Ann	Ben	Clara	Dongmei	Emily	Frank
my emails	2	190	127	25	119	5	1	6
your emails	190	3	172	35	123	7	3	5

可以看到，你向我发送了190次邮件，而我只向自己发送了2封邮件。用欧几里得距离来计算的话，在不包含me和you这两列时，我们的距离是46，包含后距离是269！因此在计算两人的距离时需要排除这个因素：

```
def distance(self, i, j):
    # 针对安然数据进行的修正
    sumSquares = 0
    for i in range(1, self.cols):
        if k != i and k != j:
            sumSquares += (self.data[k][i] - self.data[k][j]) ** 2
    return math.sqrt(sumSquares)
```

得到的层次聚类结果是：



我还用k-means++算法进行了聚类，结果是：

```
Class 5
========
chris.germany@enron.com
scott.neal@enron.com
marie.heard@enron.com
leslie.hansen@enron.com
mike.carson@enron.com

Class 6
========
sara.shackleton@enron.com
mark.taylor@enron.com
susan.scott@enron.com

Class 7
========
tana.jones@enron.com
louise.kitchen@enron.com
mike.grigsby@enron.com
david.forster@enron.com
m.presto@enron.com
```

这些结果很有趣，比如分类5中大都是贸易人员，分类7中则多是管理层。



> 安然数据中还能挖掘出很多有趣的模式，去下载完整的数据集进行尝试吧！
>
> 你也可以对其它数据集进行聚类，看看是否有新的发现。
>
> 最后，恭喜完成第八章的学习！